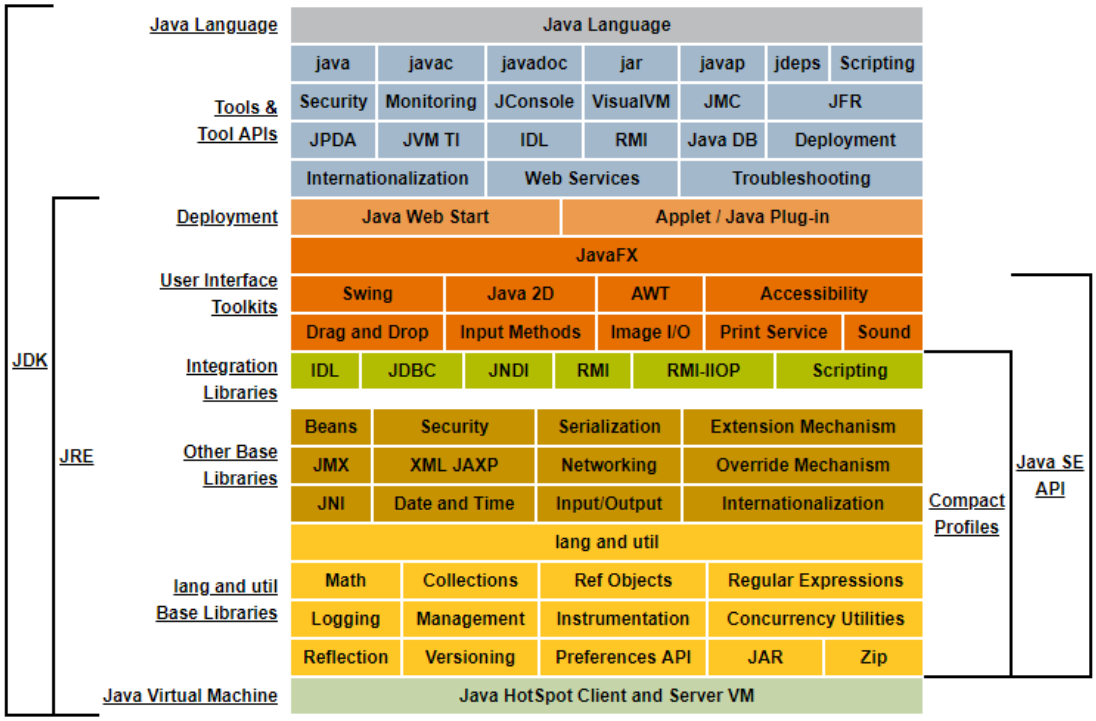
JVM 模块
JVM 体系结构

1. JVM 简介

1.1. 如何理解 JVM 呢?

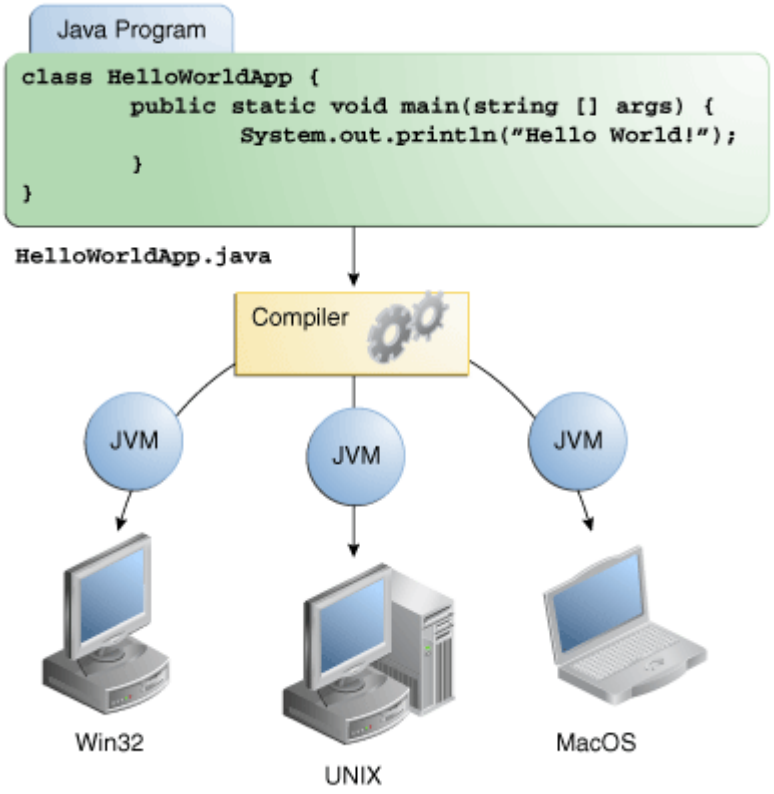
JVM(Java Virtual Machine)是:

1) 是 JAVA 平台的一部分，是一种能够运行 Java bytecode 的虚拟机。



2) 是硬件计算机的抽象(虚构)实现，可以解释执行 JAVA 字节码。

3) 是实现 JAVA 跨平台运行的基石。



1.2. 市场主流 JVM 分析?

JVM 是一种规范基于这种规范，不同公司就对此规范做了具体实现，例如市场上的一些主流 JVM 如下：

1. JRockit VM （BEA 公司研发，后在 2008 年由 Oracle 公司收购）
2. HotSpot VM （Sun 公司研发，后在 2010 年由 Oracle 公司收购）
3. J9 VM （IBM 内部使用）

说明：HotSpot 目前是应用最官方，最主要的一款 JVM 虚拟机

1.3. 为什么要学习 JVM?

深入理解 JVM 可以帮助我们从平台角度提高解决问题的能力，例如：

1）有效防止内存泄漏（Memory leak）
2）优化线程锁的使用（Thread Lock）

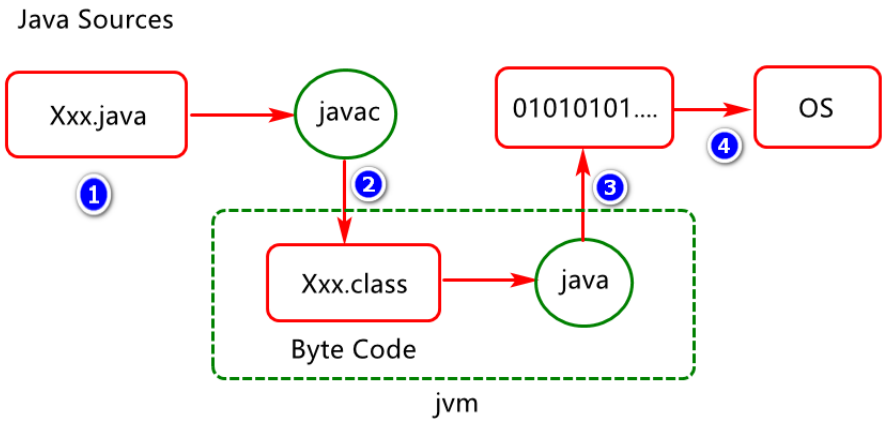
3）科学进行垃圾回收（Garbage collection）

4）提高系统吞吐量（throughput）

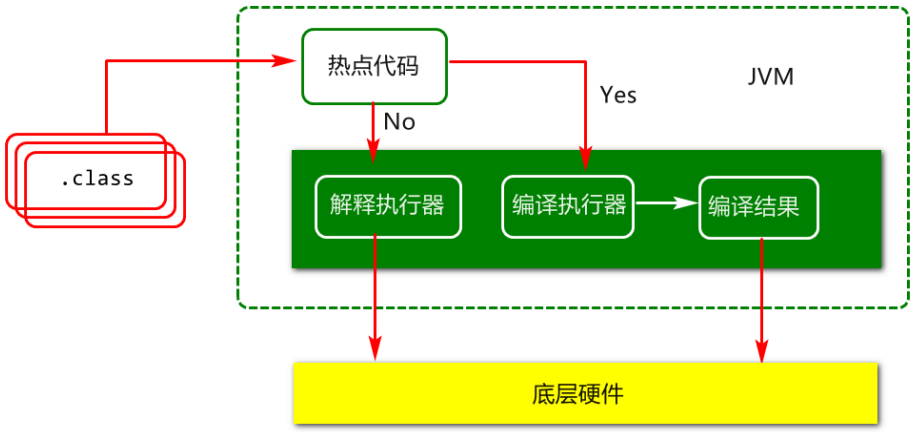
5）降低延迟(Delay)，提高其性能(performance)

1.4. 字节码底层是如何执行呢?

JAVA 源程序编译，执行过程如下图所示：



在主流的 JVM（例如 HotSpot）中,将 class 文件翻译成机器码执行时提供了两种方式，如下图所示：



其中：

1）热点代码： 一般泛指循环或高频使用的方法。

2）解释执行器：负责逐条将字节码翻译成机器码并执行。

3）编译执行器：负责即时编译(Just-In-Time compilation,JIT)执行。

如何理解 JIT 呢?

JIT(即时编译)是用来提高 java 程序运行效率的一种技术,基于这种技术,可以字节码编译成平台相关的原生机器码,并进行各个层次的优化,这些机器码会被缓存起来,以备下次使用。

为什么 JVM 中解释执行与编译执行的并存(混合模式)?

解释器与编译器两者各有优势,当程序需要迅速启动和执行的时候,解释器可以首先发挥作用,省去编译的时间,立即执行。在程序运行后,随着时间的推移,即时编译(JIT)逐渐发挥作用,它可以对反复执行的热点代码以方法为单位进行即时编译,可以获取更高的执行效率。但是如果 JIT 对每条字节码都进行编译,缓存(缓存的指令是有限的),会增加开销。所以当程序运行环境中内存资源限制较大(如部分嵌入式系统中),可以使用解释器执行节约内存,反之可以使用编译执行来提升效率。

1.5. 如何理解 JVM 的运行模式?

JVM 有两种运行模式 Server 与 Client。两种模式的区别在于,Client 模式启动速度较快,Server 模式启动较慢;但是启动进入稳定期之后 Server 模式的程序运行速度比 Client 要快很多。这是因为 Server 模式启动的 JVM 采用的是重量级的虚拟机,对程序采用了更多的优化;而 Client 模式启动的 JVM 采用的是轻量级的虚拟机。所以 Server 启动慢,但稳定后速度比 Client 远远要快。

FAQ:如何查看现在 JVM 的工作模式?

```
C:\Users\qilei>java -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
```

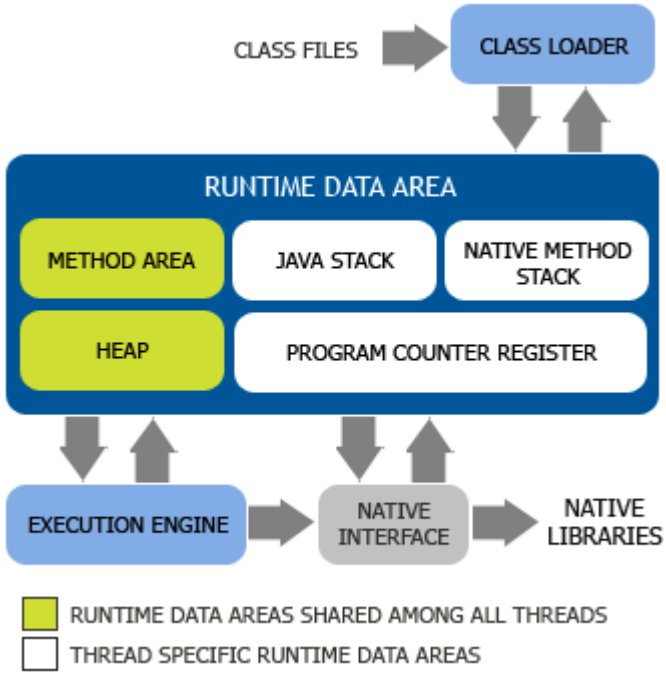
说明:现在 64 位的 jdk 中默认都是 server 模式。当虚拟机运行在-client 模式的时候,使用的是一个代号为 C1 的轻量级编译器, 而-server 模式启动的虚拟机采用相对重量级,代号为 C2 的编译器.c1、c2 都是 JIT 编译器, C2 比

C1 编译器编译的相对彻底,服务起来之后,性能更高.

2.JVM 体系结构

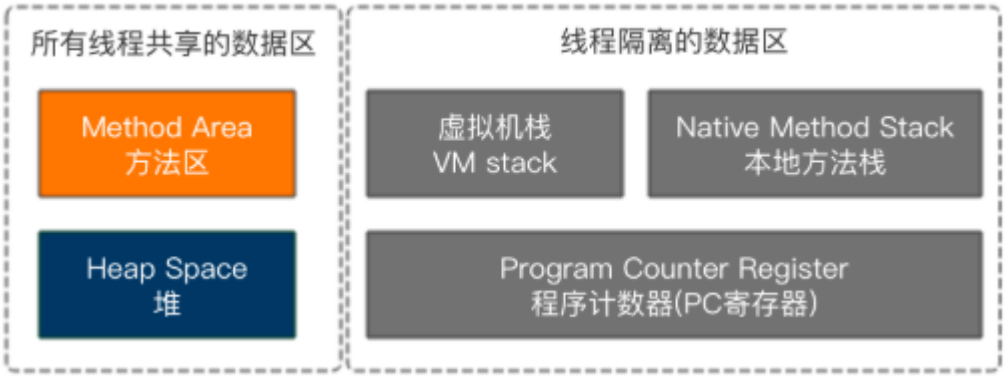
2.1. JVM 的产品架构是怎样的?

JVM (Java Hotspot Architecture: 主要分为三大部分:
1) 类加载系统 (ClassLoader System) : 负责加载类到内存
2) 运行时数据区 (Runtime Data Area) : 负责存储数据信息
3) 执行引擎 (Execution Engine) :负责调用对象执行业务

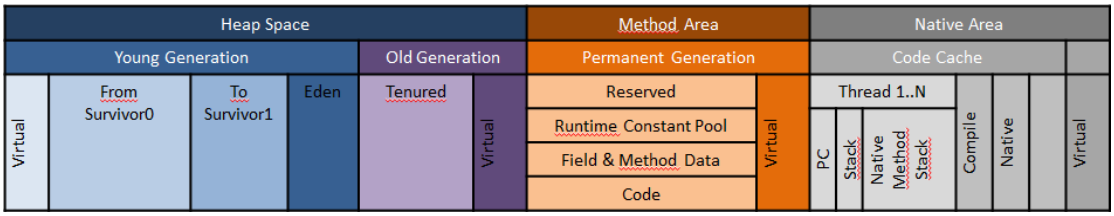


2.2. JVM 运行时内存结构是怎样的?

JVM 启动运行 Class 文件时会对 JVM 内存进行切分，我们可以将其分为线程共享区和线程独享区。如下图所示.



其运行时内存详细架构如下：



说明在 JDK8 中持久代(Permanent Generation)部分数据移到了元数据区（Metaspace），在 JDK8 中已经没有持久代。元空间的本质和永久代类似，都是对 JVM 规范中方法区的实现，不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制，但可以通过以下参数来指定元空间的大小。

2.2.1. JVM 线程共享区应用分析

1．Heap（堆内存）：

堆内存概要：

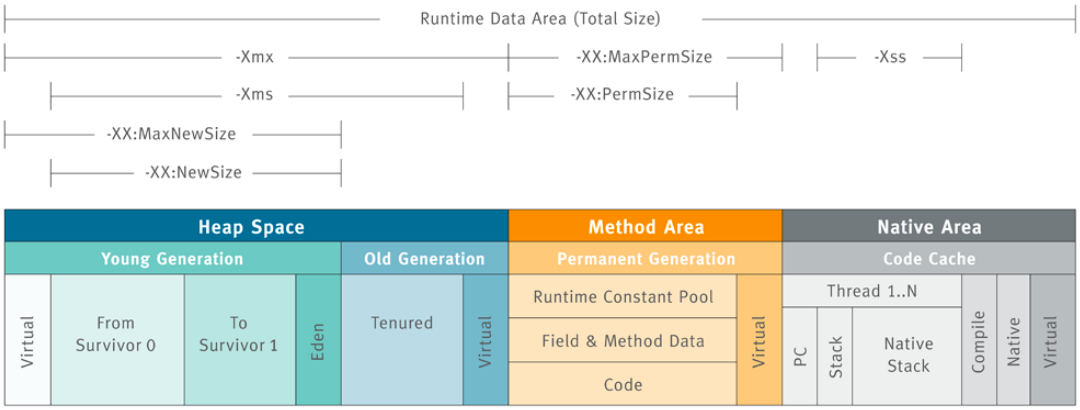
1） 虚拟机启动时创建，被所有线程共享，用于存放所有 java 对象实例.
2） 可分为年轻代(eden)和老年代(Tenured)。
3） 是垃圾收集器（GC）管理的主要区域
4） 堆中没有内存分配时将会抛出 OutOfMemoryError

对象内存分配说明：

新创建的对象一般都分配在年轻代，当对象比较大时年轻代没有足够空间还可直接分配到老年代。有时候系统为了减少 GC 开销对于小对象且没有逃逸的对象还可以直接在栈上分配。

堆内存大小配置参数说明：

1) -Xms 设置堆的最小空间大小。
2) -Xmx 设置堆的最大空间大小。
3) -XX:NewSize 设置新生代最小空间大小。
4) -XX:MaxNewSize 设置新生代最大空间大小。
5) -XX:NewRatio 新生代和老年代的比值，值为 4 则表示新生代:比老年代 1:4
6) -XX:SurivorRatio 表示 Survivor 和 eden 的比值，值为 8 表示两个
7) survivor:eden=2:8.
8) -Xss:设置每个线程的堆栈大小。



2. Method Area（方法区）

方法区概要：

1) 非堆内存，逻辑上的定义，用于存储类的数据结构信息。
2) 不同 jdk，方法区的实现不同，JDK8 中的方法区对应的是 Metaspace，是一块本地内存。

方法存储说明：

方法区内存配置参数说明：

1） -XX:PermSize 设置永久代最小空间大小（JDK8 已弃用）。

2） -XX:MaxPermSize 设置永久代最大空间大小（JDK8 已弃用）。

3） -XX:MetaspaceSize 设置元数据区最小空间。（JDK8）

4） -XX:MaxMetaspaceSize 设置元数据区最大空间。（JDK8）

2.2.2. JVM 线程私有区应用分析

1．Program Counter Register （程序计数器）

1）线程启动时创建，线程私有。

2）用于记录当前正在执行的虚拟机字节码指令地址

3）Java 虚拟机规范唯一一个没有内存溢出的区域。

2．Stack Area（虚拟机栈区）

虚拟机栈概要：

1）用于存储栈帧(Stack Frame)对象，保存方法的局部变量表、操作数栈、执行运行时常量池的引用和一些额外的附加信息。

2）一次方法调用都会创建一个新的栈帧，并压栈。当方法执行完毕之后，便会将栈帧出栈。

3）栈上分配：对于小对象（一般几十个 bytes），在没有逃逸的情况下，可以直接分配在栈上（直接分配在栈上，可以自动回收，减轻 GC 压力）；大对象或者逃逸对象无法栈上分配

说明：方法在进行递归调用时容器出现栈内存溢出。

虚拟机栈参数说明：

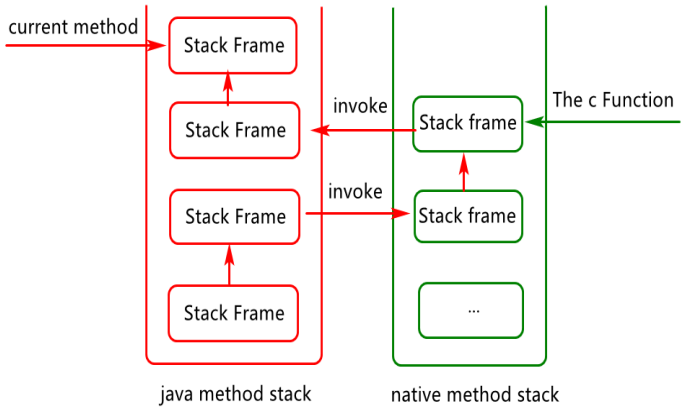
1） -Xss128k： 设置每个线程的堆栈大小

JDK5.0 以后每个线程堆栈大小为 1M，以前每个线程堆栈大小为 256K。更具应用的线程所需内存大小进行调整。在相同物理内 存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经

验值在 3000~5000 左右。

3. Native Stack Area（本地方法栈）

1）为虚拟机使用到的 Native 方法提供服务
2）用于存储本地方法执行时的一些变量信息，操作数信息，结果信息。



3. JVM 应用参数分析

3.1. 如何理解 JVM 中的内存溢出?

每个 Java 程序运行时都会使用一定量的内存， 而 JVM 内存不足以满足当前 java 程序运行时所需要的内存资源时就会出现内存溢出的现象。
代码演示：堆内存溢出

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at cgb.java.jvm.oom.TestOOM01.main(TestOOM01.java:20)
```

在如下代码中假如不断的向 list 集合中存储新的对象，list 集合底层也会不断的进行扩容，当内存中没有足够连续内存 空间存储这些对象时就会出现堆内存溢出。

```
public class TestOOM01 {
```

```
    public static void main(String[] args) {
        long t1=System.currentTimeMillis();
        try {
        List<byte[]> list=new ArrayList<>();
        for(int i=0;i<100;i++) {
            list.add(new byte[1024*1024]);
        }
        }finally {
        long t2=System.currentTimeMillis();
        //System.out.println("oom:"+(t2-t1));
        }
    }
}
```

运行时可设置虚拟机参数

```
-Xmx50m -Xms10m -XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=e:/a.dump
```

说明: 如上代码在运行了一段时间以后会出现堆内存溢出(OutOfMemoryError), 可通过调整堆内存大小, 延迟内存溢出的时间。

代码演示: 元数据内存溢出(JDK8)
当在有限的元数据内存区不断的加载新的类时会导致元数据区空间不足从而出现内存溢出现象, 例如:

```
Caused by: java.lang.OutOfMemoryError: Metaspace
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(Unknown Source)
        ... 11 more
```

```
public class TestCglibProxy {
     public static void main(String[] args) {
          while (true) {
              Enhancer enhancer = new Enhancer();
              enhancer.setSuperclass(ProxyObject.class);
              enhancer.setUseCache(false);
              enhancer.setCallback(new MethodInterceptor() {
                  @Override
                  public Object intercept(Object o,
                   Method method, Object[] os, MethodProxy proxy)
                   throws Throwable {
                      System.out.println("I am proxy");
                      return proxy.invokeSuper(o,os);
```

```
                    }
                });
                ProxyObject proxy = (ProxyObject) enhancer.create();
                proxy.greet();
            }
        }
        static class ProxyObject {
            public String greet() {
                return "Thanks for you";
            }
        }
}
```

运行时可设置虚拟机参数为：
-XX:MetaspaceSize=10M -XX:MaxMetaspaceSize=10M

3.2. JVM 工具应用分析篇

3.2.1. 命令行工具篇

---

1. Jps [options] [hostid] （hostid 为 ip 或域名地址）

jps 是用于查看有权访问的 hotspot 虚拟机的进程，当未指定 hostid 时，默认查看本机 jvm 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

例如：
代码示例

```
public class TestCommand {
    public static void main(String[] args) {
         while(true) {
         byte[] b1=new byte[1024*1024];
         }
    }
```

```
}
```

执行:

1)Jps -l 输出应用程序主类完整 package 名称或 jar 完整名称.

2)Jps -v 列出 jvm 参数,

2. jmap -heap pid

用于打印指定 Java 进程的对象内存映射或堆内存细节。

代码案例

```
public class TestCommand {
    public static void main(String[] args) {
         while(true) {
         byte[] b1=new byte[1024*1024];
        }
    }
}
```

例如获取如上代码的进程 id 之后，执行 jmap -heap 24966 可检测堆的配置信息，还可以借助 jmap -histo:live 1963 快速定位内存泄露。

其中:

1) MaxHeapFreeRatio:最大空闲堆内存比例

最大空闲对内存比例，GC 后如果发现空闲堆内存大于整个预估堆内存的 N%(百分比)，JVM 则会收缩堆内存，但不能小于-Xms 指定的最小堆的限制。

2) MinHeapFreeRatio: 最小空闲堆内存比例

GC 后如果发现空闲堆内存小于整个预估堆内存的 N%(百分比)， 则 JVM 会增大堆内存，但不能超过-Xmx 指定的最大堆内存限制。

3) **MaxHeapSize:** 即-Xmx，堆内存大小的上限

4) **InitialHeapSize:** 即-Xms，堆内存大小的初始值

5) **NewSize:**新生代预估堆内存占用的默认值

6) **MaxNewSize:** 新生代占整个堆内存的最大值

7) **OldSize:** 老年代的默认大小，

8) NewRatio:

老年代对比新生代的空间大小，比如 2 代表老年代空间是新生代的两倍大小.

9) SurvivorRatio:

Eden/Survivor 的值. 例如 8 表示 Survivor:Eden=1:8， 因为 survivor 区

有 2 个，所以 Eden 的占比为 8/10.

10)MetaspaceSize:
分配给类元数据空间的初始大小(Oracle 逻辑存储上的初始高水位，the initial high-water-mark). 此值为估计值. MetaspaceSize 设置得过大会延长垃圾回收时间. 垃圾回收过后，引起下一次垃圾回收的类元数据空间的大小可能会变大

11)MaxMetaspaceSize:
是分配给类元数据空间的最大值，超过此值就会触发 Full GC. 此值仅受限于系统内存的大小，JVM 会动态地改变此值

12)CompressedClassSpaceSize:
类指针压缩空间大小，默认为 1G.

13)G1HeapRegionSize:
G1 区块的大小，取值为 1M 至 32M. 其取值是要根据最小 Heap 大小划分出 2048 个区块.

说明:jmap 在系统调优时通常会结合 jhat 来分析 jmap 生成的 dump 文件。

3. jstack 用于生成 java 虚拟机当前时刻的线程快照

生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。
示例代码:

```
class SyncTask02 implements Runnable{
    private List<Integer> from;
    private List<Integer> to;
    private Integer target;
    public SyncTask02(List<Integer> from,List<Integer> to,Integer target) {
        this.from=from;
        this.to=to;
        this.target=target;
    }
    @Override
    public void run() {
        moveListItem(from, to, target);
```

```
    }
     private static void moveListItem (List<Integer> from,
                   List<Integer> to, Integer item) {
       log("attempting lock for list", from);
       synchronized (from) {
          log("lock acquired for list", from);
           try {
               TimeUnit.SECONDS.sleep(1);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
            log("attempting lock for list ", to);
           synchronized (to) {
              log("lock acquired for list", to);
               if(from.remove(item)){
                 to.add(item);
               }
               log("moved item to list ", to);
           }
       }
    }
    private static void log (String msg, Object target) {
        System.out.println(Thread.currentThread().getName() +
                                          ": " + msg + " " +
                            System.identityHashCode(target));
    }
}
```

```
public class TestDeadLock02 {

     public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<>(Arrays.asList(2, 4, 6, 8,
10));
        List<Integer> list2 = new ArrayList<>(Arrays.asList(1, 3, 7, 9,
11));

        Thread thread1 = new Thread(new SyncTask02(list1, list2, 2));
        Thread thread2 = new Thread(new SyncTask02(list2, list1, 9));

        thread1.start();
        thread2.start();
    }
}
```

例如 执行 jstack -l 22345 ，检测到死锁问题：

```
Found one Java-level deadlock:
=============================
"t2":
  waiting to lock monitor 0x0000000002a7ab38 (object 0x000000076b6b5c28, a java.lang.Object),
  which is held by "t1"
"t1":
  waiting to lock monitor 0x0000000002a782a8 (object 0x000000076b6b5c38, a java.lang.Object),
  which is held by "t2"

Java stack information for the threads listed above:
===================================================
"t2":
        at cgb.java.juc.lock.dead.SyncThread.run(TestDeadLock01.java:15)
        - waiting to lock <0x000000076b6b5c28> (a java.lang.Object)
        - locked <0x000000076b6b5c38> (a java.lang.Object)
        at java.lang.Thread.run(Unknown Source)
"t1":
        at cgb.java.juc.lock.dead.SyncThread.run(TestDeadLock01.java:15)
        - waiting to lock <0x000000076b6b5c38> (a java.lang.Object)
        - locked <0x000000076b6b5c28> (a java.lang.Object)
        at java.lang.Thread.run(Unknown Source)

Found 1 deadlock.
```

3.2.2. GUI 工具篇：

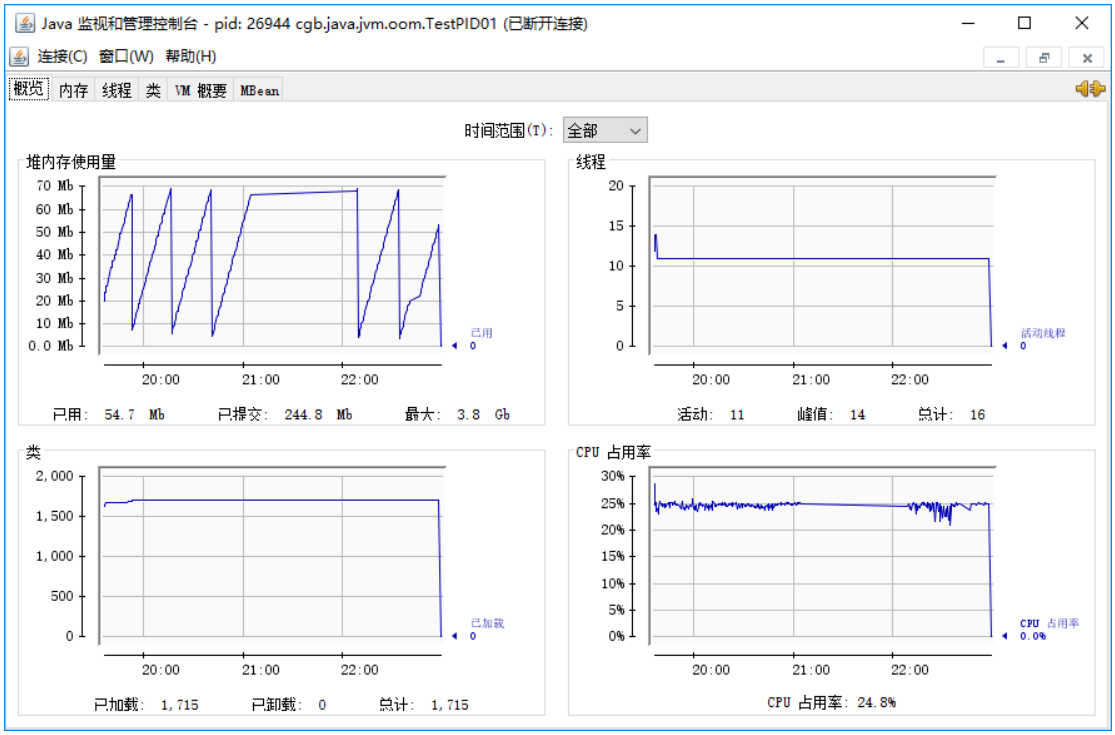
1) Jconsole （JDK 自带）

JConsole 是一个内置 Java 性能分析器，可以从命令行（直接输入 jconsole）或在 GUI shell （jdk\bin 下打开）中运行。

它用于对 JVM 中内存，线程和类等的监控。它可以监控本地的 jvm，也可以监控远程的 jvm，也可以同时监控几个 jvm。

这款工具的好处在于，占用系统资源少，而且结合 Jstat，可以有效监控到 java 内存的变动情况，以及引起变动的原因。

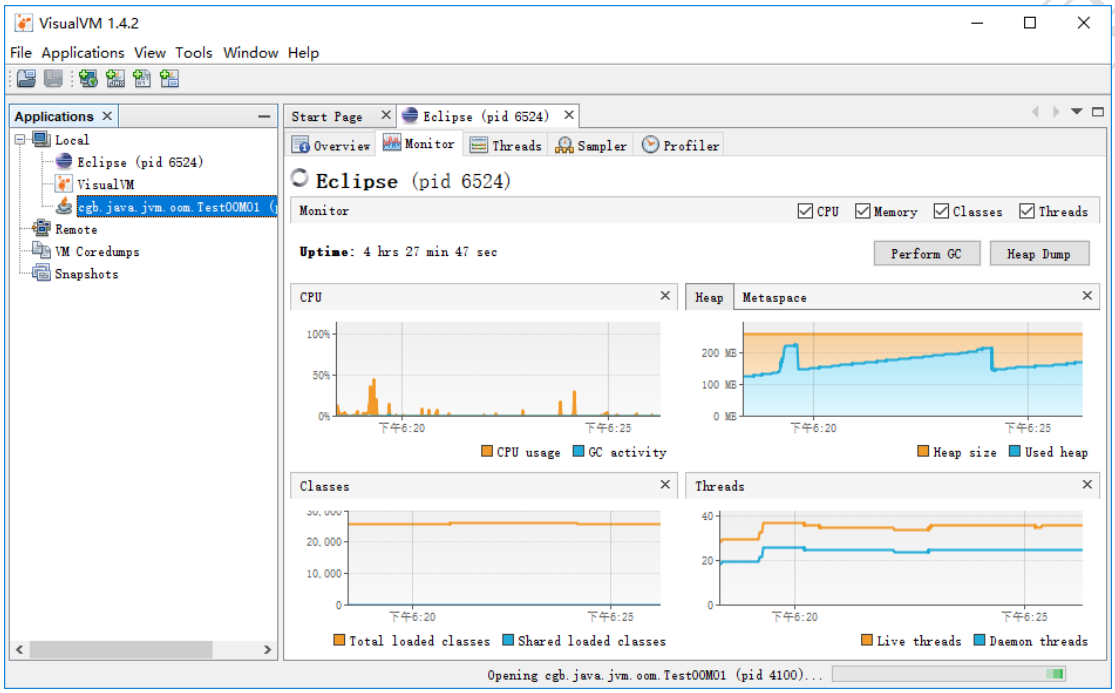
在项目追踪内存泄露问题时，很实用。



2）VisualVM (https://visualvm.github.io/download.html)

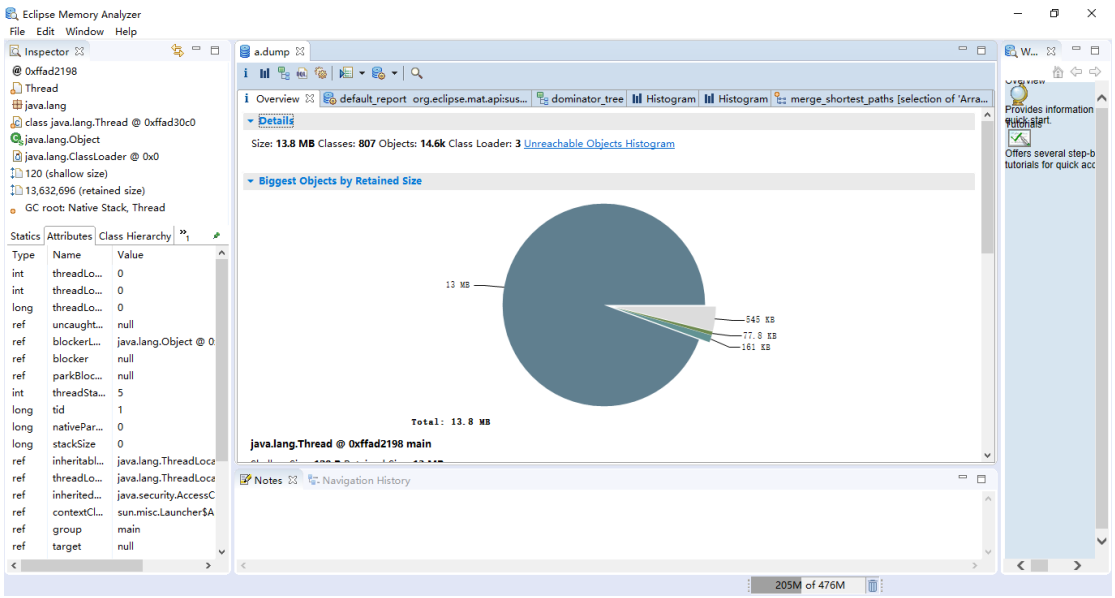
VisualVM 是一种集成了多个 JDK 命令行工具的可视化工具，它能为您提供强大的分析能力，在 VisualVM 的图形用户界面中，

您可以方便、快捷地查看多个 Java 应用程序的相关信息。



3）基于 MAT 分析 （http://www.eclipse.org/mat/）

Memory Analyzer Tool（MAT）是一个强大的基于的内存分析工具，可以帮助我们找到内存泄露，减少内存消耗。



MAT 工具的下载地址为：http://www.eclipse.org/mat/downloads.php

4. JVM 实践应用分析

4.1. 对象内存分配及日志分析

程序中大部分新创建的对象会分配在年轻代内存，例如现有如下代码

案例 1：

```
public class TestMemory01 {
    public static void main(String[] args) {
        byte[] b1=new byte[1024*1024];//分配 1MB 堆空间，考察堆空间的使用情况
```

```
        byte[] b2=new byte[1024*1024];
        byte[] b3=new byte[1024*1024];
        byte[] b4=new byte[1024*1024];
    }
}
```

配置相同内存，运行参数设置为：

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xmx1G -Xms1G -Xmn500M
```

```
    Heap
     PSYoungGen      total 448000K, used 32768K
[0x00000000e0c00000, 0x0000000100000000, 0x0000000100000000)
      eden space 384000K, 8% used
[0x00000000e0c00000,0x00000000e2c001d8,0x00000000f8300000)
      from space 64000K, 0% used
[0x00000000fc180000,0x00000000fc180000,0x0000000100000000)
      to   space 64000K, 0% used
[0x00000000f8300000,0x00000000f8300000,0x00000000fc180000)
     ParOldGen       total 536576K, used 0K [0x00000000c0000000,
0x00000000e0c00000, 0x00000000e0c00000)
      object space 536576K, 0% used
[0x00000000c0000000,0x00000000c0000000,0x00000000e0c00000)
     Metaspace       used 4774K, capacity 4930K, committed
5248K, reserved 1056768K
      class space    used 516K, capacity 561K, committed 640K,
reserved 1048576K
```

结果分析：

1）整个运行过程没有触发 GC 操作

2）对象都被分配在了年轻代的伊甸园区

配置相同内存，运行参数设置为：

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xmx20M -Xms20M -Xmn4M
```

```
    0.288: [GC (Allocation Failure) [PSYoungGen:
3072K->504K(3584K)] 3072K->1045K(19968K), 0.0015598 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
    0.541: [GC (Allocation Failure) [PSYoungGen:
3396K->504K(3584K)] 3938K->2381K(19968K), 0.0014763 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
    0.543: [GC (Allocation Failure) [PSYoungGen:
2612K->504K(3584K)] 4490K->4469K(19968K), 0.0017705 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
    Heap
```

```
    PSYoungGen      total 3584K, used 1589K
[0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)
     eden space 3072K, 35% used
[0x00000000ffc00000,0x00000000ffd0f748,0x00000000fff00000)
     from space 512K, 98% used
[0x00000000fff00000,0x00000000fff7e010,0x00000000fff80000)
     to   space 512K, 0% used
[0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
    ParOldGen       total 16384K, used 3965K
[0x00000000fec00000, 0x00000000ffc00000, 0x00000000ffc00000)
     object space 16384K, 24% used
[0x00000000fec00000,0x00000000fefdf7d8,0x00000000ffc00000)
    Metaspace       used 4773K, capacity 4930K, committed
5248K, reserved 1056768K
     class space    used 515K, capacity 561K, committed 640K,
reserved 1048576K
```

结果分析：

JVM 启动之后 288ms，共创建了 3072K KB 的对象。 第一次 Minor GC(小型 GC）完成后，年轻代中还有 504K 的对象存活，

括号中的 3584 表示新生代总大小，3072 表示 GC 前堆大小，1045 表示 GC 后堆大小，括号中的 19968k 为总堆大小。0.0015598 secs 表示 GC 时间。

修改配置参数

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps   -Xmx20M -Xms20M -
Xmn10M
```

```
   0.467: [GC (Allocation Failure) [PSYoungGen:
7276K->1016K(9216K)] 7276K->3367K(19456K), 0.0018520 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
   Heap
    PSYoungGen      total 9216K, used 3304K
[0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)
     eden space 8192K, 27% used
[0x00000000ff600000,0x00000000ff83c1d0,0x00000000ffe00000)
     from space 1024K, 99% used
[0x00000000ffe00000,0x00000000ffefe010,0x00000000fff00000)
     to   space 1024K, 0% used
[0x00000000fff00000,0x00000000fff00000,0x0000000100000000)
    ParOldGen       total 10240K, used 2351K
[0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)
     object space 10240K, 22% used
[0x00000000fec00000,0x00000000fee4be08,0x00000000ff600000)
    Metaspace       used 4774K, capacity 4930K, committed
```

```
5248K, reserved 1056768K
     class space    used 516K, capacity 561K, committed 640K,
reserved 1048576K
```

4.2. Tomcat 中内存配置应用分析

tomcat 一般都有默认的内存大小，其默认值对整个物理内存来说非常小，如果不配置 tomcat 的内存，会大大浪费服务器的资源，验证影响系统的性能，所以对 tomcat 的内存配置对用户量比较大的系统尤为重要。

Windows 平台下在 bin 目录下的 catalina.bat 文件中，找到@echo off 然后再它的下面一行添加如下类似语句。

```
SET CATALINA_OPTS= -Xms2048m -Xmx2048m -Xmn500m
```

参数说明：

```
-server：一定要作为第一个参数，在多个 CPU 时性能佳
-Xms：java Heap 初始大小。 默认是物理内存的 1/64。
-Xmx：java heap 最大值。建议均设为物理内存的一半。不可超过物理内存。
-Xmn：young generation(年轻代)的 heap 大小。一般为 Xmx 的 3、4 分之一
-XX:MetaspaceSize=128m 初始元空间大小，默认一般为 21m。
-XX:MaxMetaspaceSize=256m 最大元空间大小，默认无上限，由 OS 内存决定
```