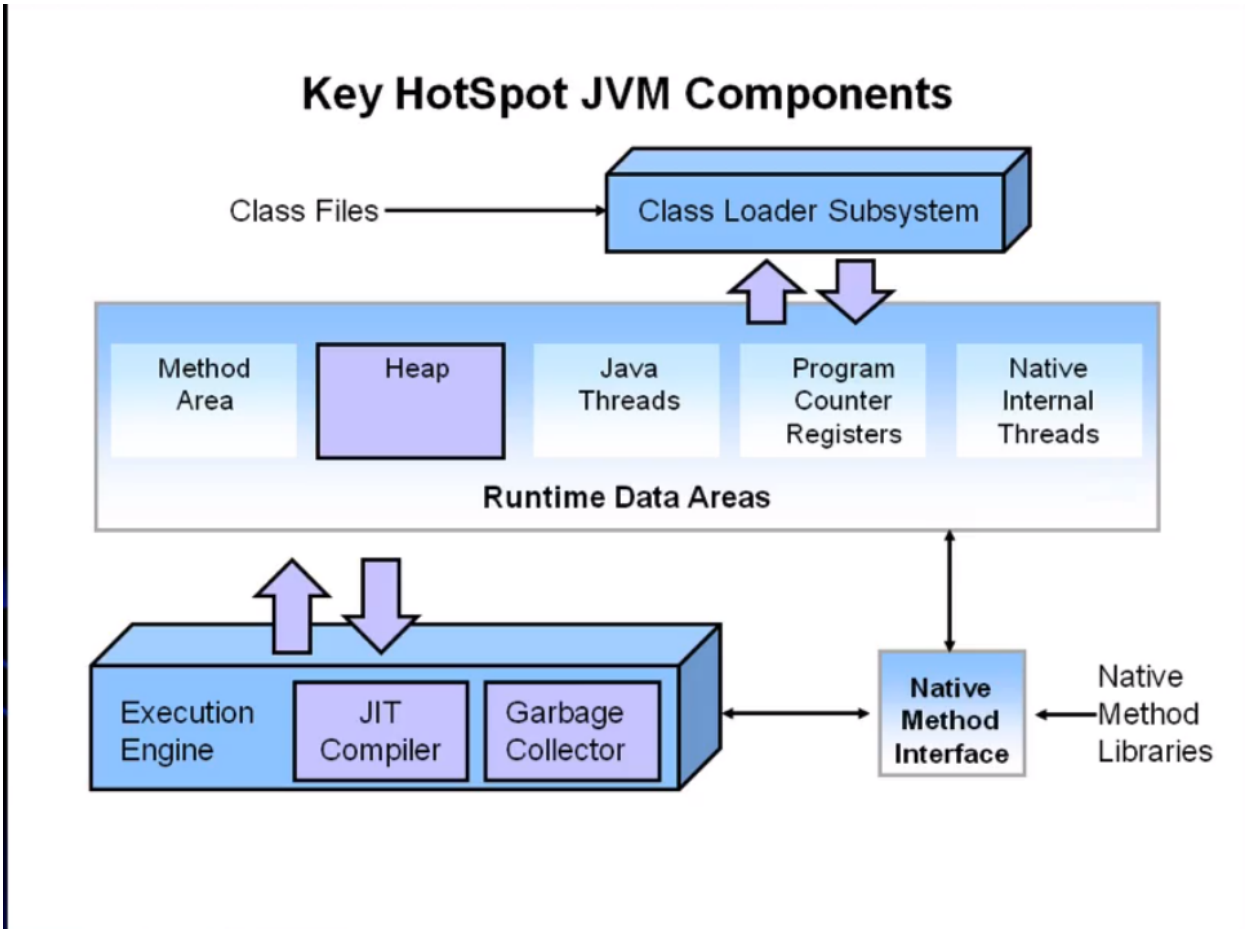
JVM 内存模型

内存结构

1. 虚拟机栈: Stack Frame 栈帧, 方法执行过程中的压栈和出栈的执行过程。
2. 程序计数器 (Program Counter), 字节码执行顺序。
3. 本地方法栈: native 来获取的JVM提供的本地方法。
4. 堆(Heap): JVM管理的最大的一块内存空间。与堆相关的是一个重要概念是垃圾收集器。现代　　几乎所有的垃圾收集器都是采取的分代收集算法， 所以对空间也是基于这一点进行了相应的划分：新生代与年老代. Eden 空间, From Survivor 空间与 To Survivor 空间。
5. 方法区(Method Area): 存储元数据信息。永久代 (Permanent Generation) 从JDK1.8开始彻底废弃永久代， 使用元空间(Meta Space)来替代。
6. 运行时常量池：方法区的一部分内容。
7. 直接内存： Direct Memory, 堆外内存，不是由JVM来管理，是通过操作系统来管理的。与Java NIO密切相关的。Java 通过DirectByteBuffer来 操作直接内存。



Java对象的创建过程

new 关键创建对象的3个步骤

1. 在堆内存中创建出对象的实例。

2. 为对象的实例成员变量赋初始值。

3. 将对象的引用返回。

指针碰撞（前提是堆中的空间通过一个指针进行分割，一侧是已经被占用的空间，另一侧是未被

占用的空间）

空闲列表（前提是堆空间中已经被使用，未被使用的是交织在一起的。这时虚拟机就需要通过一

个列表来记录哪些是可以使用的，哪些是已经被使用的， 接下来找出可以容纳新创建的

对象的未被使用的空间，再此空间存放对象， 同时还要修改列表上的记录）。

对象在内存中的布局：

1. 对象头
2. 实例数据 (即我们再一个类中声明)
3. 对齐填充 (可选)

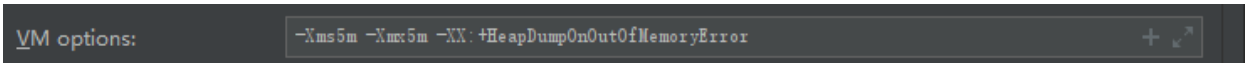
引用访问对象的方式:

1. 使用句柄的方式。
2. 使用直接指针的方式。

分析工具jvisualvm和jconsole

1. 配置JVM参数

```
-Xms2m
-Xmx2m
-XX:+HeapDumpOnOutOfMemoryError
```




```
-XX:MaxMetaspaceSize=10m  //设置元空间大小
```

Jdk1.8 元空间

元空间存储类的基本元数据,如类的层级信息,方法数据和方法信息（如字节码，栈和变量大小),运行时常量池,已确定的符号引用和虚方法表。

jps -l 获取所有java的进程号

jmap: jmap -clstats pid 打印类加载器数据

```
D:\workspace\GUI\GUI\target\classes\main\java\Test1>jmap -clstats 9488
Attaching to process ID 9488, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.131-b11
finding class loader instances ..done.
```

computing per loader stat ..done.
please wait.. computing liveness....liveness analysis may be inaccurate ...
class_loader classes bytes parent_loader alive? type

<bootstrap> 663 1228703 null live <internal>
0x0000000780c28bf0 25 69015 0x0000000780c0fb28 live
sun/misc/Launcher$AppClassLoader@0x00000007c000f6a0
0x0000000780fe8ab8 0 0 0x0000000780c28bf0 dead
java/util/ResourceBundle$RBClassLoader@0x00000007c006dd08
0x0000000780c0fb28 0 0 null live
sun/misc/Launcher$ExtClassLoader@0x00000007c000fa48

total = 4 688 1297718 N/A alive=3, dead=1 N/A

jstat: jstat -gc pid 用来打印元空间的信息

D:\workspace\GUI\GUI\target\classes\main\java\Test1>jstat -gc 9488
S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU
YGC YGCT FGC FGCT GCT
7680.0 7680.0 0.0 0.0 49152.0 4925.2 130048.0 0.0 4480.0 770.4 384.0
75.9 0 0.000 0 0.000 0.000

jcmd（从jdk1.7开始新增加的命令）

1. jcmd pid VM.flags 查看jvm的启动参数

9488:
-XX:CICompilerCount=3 -XX:InitialHeapSize=199229440 -
XX:MaxHeapSize=3185573888 -XX:MaxNewSize=1061683200 -
XX:MinHeapDeltaBytes=524288 -XX:NewSize=66060288 -XX:OldSize=133169152 -
XX:+UseCompressedClassPointe
rs -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:-
UseLargePagesIndividualAllocation -XX:+UseParallelGC

2. jcmd pid help 查看当前可用命令
3. jcmd pid help JFR.dump 查看具体命令的选项
4. jcmd pid PerfCounter.print 查看JVM性能相关的参数
5. jcmd pid VM.uptime 查看类的启动时长
6. jcmd pid GC.class_histogram: 查看类的统计信息
7. jcmd pid Thread.print: 查看线程的堆栈信息
8. jcmd pid GC.heap_dump filename: 导出Heap Dump文件, 导出的文件可以通过 jvisualvm 查看
9. jcmd pid VM.system_properties: 查看JVM的属性
10. jcmd pid VM.version: 查看JVM进程的版本信息

11. jcmd pid VM.command_line: 查看JVM启动的命令行参数信息

Jstack:查看或者导出Java进程中的堆栈信息

jmc:Java Mission Control

jhat: 分析堆转储信息，在jdk9 以后已被移除. 移除原因:

https://www.infoq.com/news/2015/12/OpenJDK-9-removal-of-HPROF-jhat/

JVM内存溢出分析场景

1. 堆溢出, 如果不断的创建对象，那么在对象的数量到达最大堆的容量后就会产生堆溢出

2. 虚拟机栈和本地方法栈溢出。

- 如果栈的深度大于虚拟机允许的最大深度。则抛出 StackOverflowError异常。
- 如果虚拟机在拓展栈的时候，无法申请到足够内存。则抛出OutOfMemoryError异常。

1. 方法区和运行时常量池溢出。

- 如果在运行时不断地创建大量的类最会导致方法区溢出。

1. 本机直接内存溢出，可以通过反射Unsafe实例来分配直接内存或通过DirectByteBuffer类