Java Virtual Machine 是Java平台的基石，包含相应的技术规范、实现（JVM的实现就是JRE）、运行实例 为了实现一次编写到处运行 (Write Once Run Anywhere)的理念，JVM是和平台相关的

Java HotSpot Client VM (client VM) ：主要用于减少启动时间和内存占用，启动应用时使用-client指定 Java HotSpot Server VM (server VM)：用于最大化的执行速度，启动应用时使用-server指定

JVM主要包括三个子系统

* 类加载子系统 Class Loader
* 运行时数据区 Runtime Data Area
* 执行引擎 Execution Engine

img

img

JVM构架

## 运行时数据区

JVM定义了不同的运行时数据区，一些是在JVM启动时就创建的，有些数据区的生命周期和线程有关，主要有5种

### 程序计数器 Progarm Counter Registers

JVM支持许多线程同时运行，每个JVM线程都有自己的pc寄存器 当线程运行的是Java方法时，存储当前正在执行的JVM指令地址 如果是Native方法，未定义

### Java虚拟机栈 Java Virtual Machine Stacks

每个线程都有一个私有的Java虚拟机栈，和线程的生命周期相同。描述了Java方法执行的内存模型 用来存储栈帧（Frame）

**Frame** 每个方法在调用的时候，都会创建一个栈帧，包含局部变量（Local Variables）、操作数栈（Operand Stacks）、动态链接（Dynamic Linking）、方法的调用者信息（Normal/Abrupt Method Invocation Completion） **局部变量** 存储boolen,byte,char,short,int,long,float,double,reference（引用）,returnAddress(指向一条操作码) 局部变量的数组长度是在编译时确定的，之后不再改变 同样用于方法调用中的参数传递，0号索引用于存储this **操作数栈** 深度由编译时确定 加载常量，局部变量或者字段到操作数栈，出栈计算出结果，压栈 还可以用来准备传递到函数中的参数，接收方法的返回值

和常见的语言（例如C）的栈类似：存储局部变量，局部结果，用于方法调用和返回等

**Exception** StackOverflowError:线程请求的栈深度大于JVM的的允许范围（函数调用层级过多导致） OutOfMemoryError:内存不足，无法动态扩展内存时

-Xss<size>:设定线程栈的大小 (默认单位是byte,支持 K/k M/m G/g 作为单位)

### 本地方法栈 Native Method Stack

JVM使用到Native方法的时候使用本地方法栈 每一个线程,将创建一个单独的本地方法栈。

和Java虚拟机栈的异常相同

### 堆Heap

被**所有的JVM线程共享**，所有的类对象和数组都要在堆上分配 堆是虚拟机所管理的内存中最大的一块，也是Garbage Collector管理的主要区域 Java堆的内部可以不连续，堆空间大小可以是固定的，也可以是可扩展收缩的

Exception: OutOfMemoryError: 请求更多的堆空间无法满足

-Xms<size>:初始堆大小  
-Xmx<size>:最大堆大小

### 方法区 Method Area

被所有Java线程共享，存储被虚拟机加载的类信息，例如运行时常量池，字段和方法数据，方法的代码，构造函数等，JVM规范没有限定方法区的位置，可以是固定大小或者可扩展，内存不要求连续

Hotspot虚拟机中的方法区之前也叫做永久代,PermGen，Java8 后叫做元空间,Matespace，分配在本地内存Native Memory中，它的大小可以自动的增长，可以通过XX：MaxPermSize，-XX:MaxMetaspaceSize=<size>设定大小 因为是在本地内存(native memory)分配,所以其最大可利用空间是整个系统内存的可用空间，不容易遇到OutOfMemoryError错误

img

img

image.png

### 运行时常量池 Runtime Constant Pool

是方法区的一部分，类似于常规语言的符号表 类或者接口的.class文件中，有一个常量表属性(constant\_pool table)，存放各种字面常量(在编译时确认)和字段引用(在运行时确认)，加载到虚拟机时，对应加载到运行时常量池

Exception: OutOfMemoryError: 方法区无法满足内存分配需求

### 堆外内存

Native Memory或者Off-Heap内存空间，例如NIO中的DirectBuffer就是使用native函数直接分配的堆外内存，可以通过-XX:MaxDirectMemorySize来设置NIO直接缓冲区的最大值。

使用堆外内存的好处：

* 可以扩展更大的内存空间
* 能减少GC时间
* 可以在进程间共享数据

## Java堆中对象的分配、布局、访问

### 对象的创建

通过new指令创建对象时，虚拟机的执行流程

1. 类加载
2. 为对象分配内存（大小是确定的，在加载后便已经确定），两种分配策略，**指针碰撞（Bump the Pointer）**和**空闲列表（Free List）**，具体的策略取决于GC算法，具有整理功能使用碰撞指针，否则使用空闲列表。并发分配内存时，对分配内存空间的动作进行**同步处理**（基于CAS，Compare and Swap），或者采用**本地线程分配缓冲（Thread Local Allocation Buffer，TLAB）**策略，每个线程在Java堆中预先分配一块内存，在该线程的内存块内进一步分配
3. 内存空间初始化为零值
4. 对象头（Object Header）信息设置
5. 执行init方法，即构造方法

### 对象的内存布局

#### 对象头（Header）

HotSpot虚拟机的对象头(Object Header)包括两部分信息

* 存储对象自身的**运行时数据**， 如哈希码(identity\_hashcode，对象的标识)、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分数据的长度在32位和64位的虚拟机中分别为32个和64个Bits，官方称它为“Mark Word”，hotspot内部对应[markOop.hpp](http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/87ee5ee27509/src/share/vm/oops/markOop.hpp)，其中oop全拼是ordinary object pointer，表示指向一个对象的指针
* 类型指针（指向方法区的对象类型数据）

如果是数组类型，还会有4bytes(32bits)用于记录数组的长度

64为JVM对象头的结构： 其中MarkWord里的数据会随着锁标志位的变化而变化

|------------------------------------------------------------------------------------------------------------|--------------------|  
| Object Header (128 bits) | State |  
|------------------------------------------------------------------------------|-----------------------------|--------------------|  
| Mark Word (64 bits) | Klass Word (64 bits) | |  
|------------------------------------------------------------------------------|-----------------------------|--------------------|  
| unused:25 | identity\_hashcode:31 | unused:1 | age:4 | biased\_lock:1 | lock:2 | OOP to metadata object | Normal |  
|------------------------------------------------------------------------------|-----------------------------|--------------------|  
| thread:54 | epoch:2 | unused:1 | age:4 | biased\_lock:1 | lock:2 | OOP to metadata object | Biased |  
|------------------------------------------------------------------------------|-----------------------------|--------------------|  
| ptr\_to\_lock\_record:62 | lock:2 | OOP to metadata object | Lightweight Locked |  
|------------------------------------------------------------------------------|-----------------------------|--------------------|  
| ptr\_to\_heavyweight\_monitor:62 | lock:2 | OOP to metadata object | Heavyweight Locked |  
|------------------------------------------------------------------------------|-----------------------------|--------------------|  
| | lock:2 | OOP to metadata object | Marked for GC |  
|------------------------------------------------------------------------------|-----------------------------|--------------------|

#### 实例数据（Instance Data）

存储父类子类的数据，存储顺序与虚拟机字段分配策略和源码中的定义顺序有关，相同宽度的字段会被分配到一起，HotSpot默认策略是从长到短排列，引用排最后: long/double --> int/float --> short/char --> byte/boolean --> Reference

### 对齐填充（Padding）

占位符，HotSpot VM对象的起始地址必须是8字节的整数倍

#### 查看内存布局

可以使用openJDK中的[JOL (Java Object Layout)](http://openjdk.java.net/projects/code-tools/jol/)查看内存布局

$ java -version  
java version "1.8.0\_181"  
Java(TM) SE Runtime Environment (build 1.8.0\_181-b13)  
Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)  
  
$ java -jar jol-cli-0.9-full.jar internals java.lang.Object  
# WARNING: Unable to attach Serviceability Agent. You can try again with escalated privileges. Two options: a) use -Djol.tryWithSudo=true to try with sudo; b) echo 0 | sudo tee /proc/sys/kernel/yama/ptrace\_scope  
# Running 64-bit HotSpot VM.  
# Using compressed oop with 3-bit shift.  
# Using compressed klass with 3-bit shift.  
# WARNING | Compressed references base/shifts are guessed by the experiment!  
# WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.  
# WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.  
# Objects are 8 bytes aligned.  
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]  
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]  
  
Instantiated the sample instance via default constructor.  
  
java.lang.Object object internals:  
 OFFSET SIZE TYPE DESCRIPTION VALUE  
 0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000) (1)  
 4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000) (0)  
 8 4 (object header) e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)  
 12 4 (loss due to the next object alignment)  
Instance size: 16 bytes  
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

可以看到，mark work为前两行，占8字节，kclass pointer占4字节(这里为64位虚拟机，使用了指针压缩)，对齐消耗4字节

#### 指针压缩

对应的JVM选项是-XX:+UseCompressedOops默认开启，当Java堆大小小于32 GB时使用压缩指针。 此时对象引用实际表示的是32位偏移而不是64位指针，可以提高性能（通常64位JVM消耗的内存会比32位的大1.5倍）

Java堆中的指针指向的对象需要进行8字节对齐。 压缩oops表示指向64位Java堆基地址的32位对象偏移量。 因为它们是对象偏移而不是字节偏移，所以它们可以用于处理4G个对象（不是字节），或者32GB大小的堆，因此可以存储时可以右移3bit。 使用过程中，JVM将它们放大8倍并加上Java堆基址以查找它们引用的对象。使用压缩oops的对象大小与32位系统中的对象大小相当

当Java堆大小大于32G时，也可以使用压缩指针，通过-XX:ObjectAlignmentInBytes=alignment选项设定对齐字节数目，必须是2的幂次，且小于等于256，默认为8

开启指针压缩时，堆中的一下oop会被压缩：

* 每个对象的klass字段
* 每个oop实例字段
* oop数组的每个元素（objArray）

### 对象的访问定位

通过栈上的reference数据指向堆上的具体数据，有两种实现方式：

* 直接指针访问:reference中存储的直接就是对象地址，访问速度更快，HotSpot使用该方式
* 句柄访问：reference中存储的是稳定的句柄地址,对象被移动时只会改变句柄中的实例数据指针，reference本身不需要修改