# JVM

| 类加载子系统 | | | java栈 |
| --- | --- | --- | --- |
| 方法区 | Java堆 | 直接内存 | 本地方法 |
| 垃圾回收系统 | | | pc寄存器 |
| 执行引擎 | | | |

- 基本结构
  - 类加载子系统

    > 类加载系统负责从文件系统或网络中加载**Class**信息，加载的类信息存放于一块称为**方法区**的内存空间。

  - 直接内存

    > java的NIO库允许Java程序使用直接内存。直接内存是在java堆之外的，从系统申请的内存空间，访问速度快于 java堆。出于性能考虑，读写频繁的场景可以考虑使用直接内存。直接内存的大小不能通过Xmx参数来设置。

  - Java栈

    > 每个虚拟机线程都有一个Java栈，并且是私有的。Java栈会在线程创建的时候被创建。Java栈中保存这帧信息，局部变量，方法参数，同时和Java方法的调用、返回相关。

  - 方法区

    > 除了类信息外，方法区中可能还会存放运行时的常量池信息，包括字符串字面量和数字常量（这部分常量信息是CLass文件中常量池部分的内存映射）

  - java堆

    > java堆在虚拟机启动的时候建立，它是Java程序最主要的内存工作区

> 域。几乎所有java对象实例都存放在这里。 堆空间是所有线程共享的，是与Java程序密切相关的内存区间。

- pc寄存器

> pc寄存器是线程的私有空间，虚拟机会为每个线程创建pc寄存器。一个线程总是在执行一个方法，这个方法被称为当前方法，如果这个方法不是本地方法，pc寄存器会执行当前正在执行的指令，如果是本地方法，则为undefined。

- 垃圾回收系统

> 垃圾回收是虚拟机的重要组成部分，垃圾回收器可以对方法区、java堆和直接内存进行回收，其中java堆是重点。

- 本地方法

> 类似Java栈，用于调用本地方法（native）

- 执行引擎

> 虚拟机的核心组件之一，负责执行字节码。

---

- **stackoverflowError**（-Xss修改栈空间大小）

> 栈溢出的原因：
> 1.递归方法没有出口
> 2.栈帧中有局部变量表，如果方法的参数和局部变量太多的话，会造成帧中变量表膨胀，占用更多的栈空间，从而容易引起栈溢出的错误。 **写代码是要注意局部变量的作用域和在栈帧中变量表槽位的复用，节省空间。**

```java
//jclasslib可查看class文件中变量的详细信息

// a和b都是一直到函数结束了才失效，无法复用
public void localvar1(){
    int a = 0;
    System.lout.println(a);
    int b = 0;
}

//a先失效后，b可以复用a的槽位
public void localvar2(){
    {
      int a = 0;
      System.lout.println(a);
    }
      int b = 0;
}
```

```java
//局部变量表中直接或间接引用的对象都是不会被回收的。
//-XX:PrintGC参数可打印gc前后的堆大小，确定是否执行回收

//a被引用了，gc无效
public void localvarGc1(){
    byte a = new byte[60*1024*1024];
    System.gc();
}


//a设为null后取消了强引用，gc有效
public void localvarGc2(){
   byte a = new byte[60*1024*1024];
   a = null;
   System.gc();
}

//虽然a在gc前已失效，但是a还在变量表中，所以gc无效
public void localvarGc3(){
    {
         byte a = new byte[60*1024*1024];
    }
       System.gc();
}

//a失效后，c复用了a的槽位，a已被销毁，所以gc成功
public void localvarGc4(){
    {
         byte a = new byte[60*1024*1024];
    }
       int c = 0;
       System.gc();
}

//Gc1执行后，Gc1的栈帧也被销毁了，所以gc成功
public void localvarGct5(){
    localvarGc1();
    System.gc();
}
```

- 栈上分配（-XX:+DoEscapeAnalysis开启逃逸分析）

> 栈上分配是虚拟机提供的优化技术。主要是利用逃逸分析，将线程中的私有对象对象分配到栈中，不会引起大面积GC。（由于栈空间比较小，只适合小对象。）

- 方法区（-XX:MaxMetaspaceSize指定大小）

> 在JDK1.7之后的版本中，永久区被移除了，取代的是元数据区，这是一块堆外的直接内存，所以如果不指定大小，默认情况虚拟机会耗尽所有的可用系统内存。

```
Cased by: java.lang.OutOfMemoryError: Metaspce
   at ...
```

---

- 虚拟机用参数跟踪jvm日志

> **-XX:PrintGCDetails** 打印每次GC详细信息
> **-XX:PrintHeapAtGC** 打印堆GC信息
> **-Xss:** 指定线程栈大小
> **-XX:MaxDirectMemorySize** 设置直接内存大小，默认为最大堆空间（-Xmx）

```
//读写直接内存快与堆内存，但是在申请直接内存时慢于堆内存，-server参数设置为服务器模式
ByteBuffer.allocateDirect(?);//直接内存
ByteBuffer.allocate(?);//堆内存
```

- 虚拟机工作模式

> Server模式启动较慢，执行速度快于Client模式。—XX:+PrintFlagsFinal 可以查看默认给定参数。

---

- 垃圾回收

> **强引用**不会被回收
> **软引用**在堆空间不足时，会被回收
> **弱引用**被发现就会直接回收
> **虚引用**用于回收跟踪
> *软引用和弱引用适用于缓存*

-停顿现象 Stop-The-World

> 垃圾回收是会造成应用的停顿，时间成了会造成应用的卡顿或直接卡死。 通过参数设置来避免这一现象：
> 增大新生代大小减少GC，但是会增加每次GC时间 开启新生代ParNew回收器：
> -XX:+UseParNewGC cpu大于8时 —XX:ParallelGCThreads=3+((5*cpu_count)/8)
> -XX:ExplicitGCInvokeConcurrent ,让System.gc()不执行fullgc，执行当前的并行GC。

---

- Java调试命令
  - jps 显示Java进程
  - jstack 查看线程堆栈
  - jstat 查看虚拟机运行时信息

> jstat -gc（gccapacity） pid 查看gc信息
> -gccause 查看gc原因
> -gcnew 查看新生代信息
> jmp 到处堆快照
> jhat 堆快照分析工具，通过http://127.0.0.1:7000访问结果,用oql查找信息

```
jmp -histo pid >/temp/shot.txt

jhat /temp/shot.txt
```

- 性能监控工具（linux命令）
  - top
  - vmstat
  - iostat
  - pidstat
- BTrace

---

- String
  - 不变性:所有的字符串都时放在常量池，使用的只是常量的指针
  - 针对常量池的优化

```
//返回true
public static void main(String[] args) {
        String a = "abc";
        String b = "abc";
        System.out.println(a.intern() == b.intern());

    }
```

---

- **浅堆和深堆**
  - 浅堆：对象本身所占的内存
  - 深堆：对象本身和对象直接或间接调用的对象之和

---