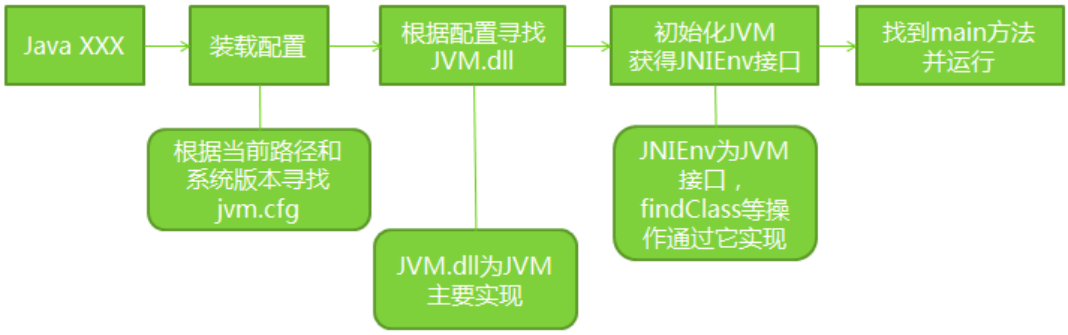
http://www.cnblogs.com/smyhvae/p/4748392.html

主要内容如下：

- JVM启动流程
- JVM基本结构
- 内存模型
- 编译和解释运行的概念

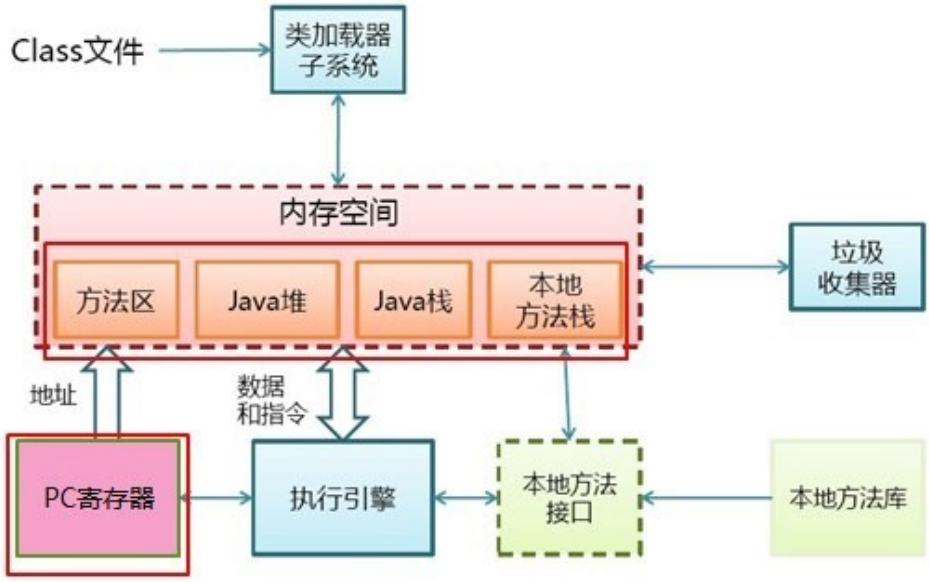
一、JVM启动流程：



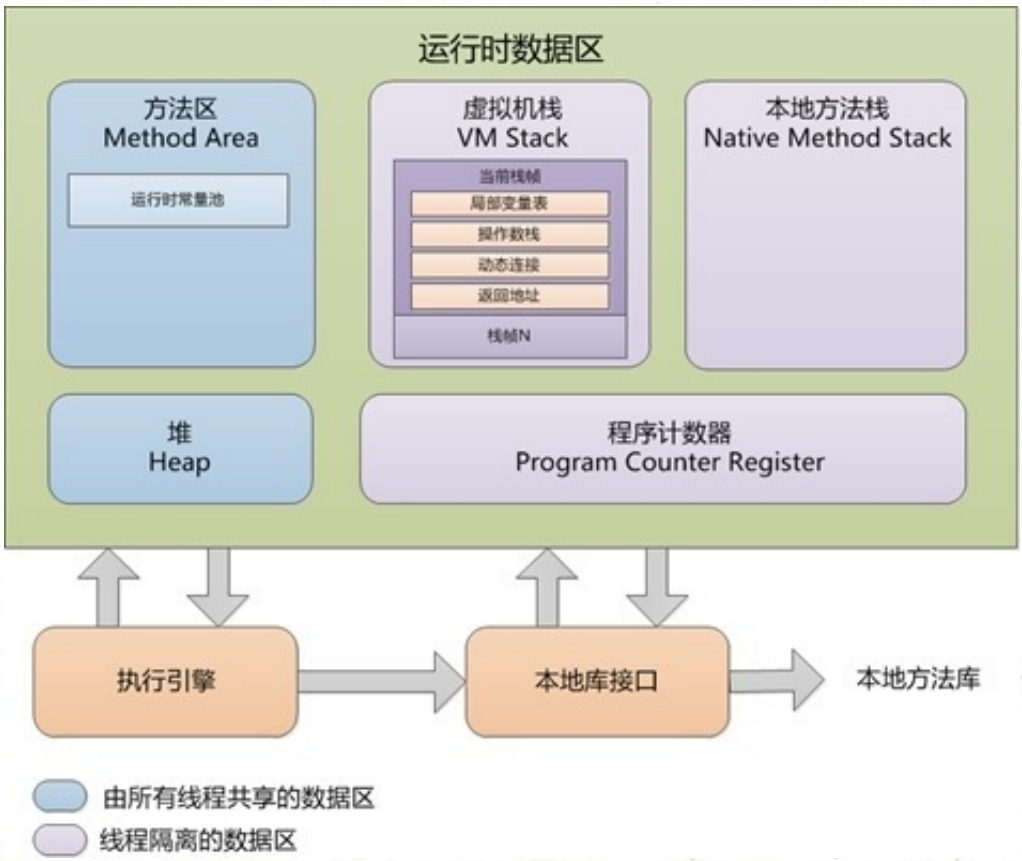
JVM启动时，是由java命令/javaw命令来启动的。

二、JVM基本结构：

JVM基本结构图：



《深入理解Java虚拟机（第二版）》中的描述是下面这个样子的：



Java中的内存分配：

Java程序在运行时，需要在内存中的分配空间。**为了提高运算效率，就对数据进行了不同空间的划分**，因为每一片区域都有特定的处理数据方式和内存管理方式。

具体划分为如下**5个内存空间**：（非常重要）

- 栈：存放局部变量
- 堆：存放所有**new出来的东西**
- 方法区：被虚拟机加载的类信息、常量、静态常量等。
- 程序计数器(和系统相关)
- 本地方法栈

1、程序计数器：

每个线程拥有一个PC寄存器

在线程创建时创建

指向下一条指令的地址

执行本地方法时，PC的值为undefined

2、方法区：

保存装载的类信息

类型的常量池

字段，方法信息

方法字节码

通常和永久区(Perm)关联在一起

3、堆内存:

和程序开发密切相关

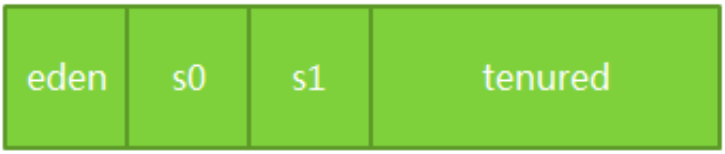
应用系统对象都保存在Java堆中

所有线程共享Java堆

对分代GC来说，堆也是分代的

GC管理的主要区域

现在的GC基本都采用分代收集算法，如果是分代的，那么堆也是分代的。如果堆是分代的，那堆空间应该是下面这个样子:



复制算法

上图是堆的基本结构，在之后的文章中再进行详解。

4、栈内存:

- 线程私有，生命周期和线程相同
- 栈由一系列帧组成（因此Java栈也叫做帧栈）
- 帧保存一个方法的局部变量、操作数栈、常量池指针
- 每一次方法调用创建一个帧，并压栈

解释:

Java虚拟机栈描述的是Java方法执行的内存模型：**每个方法被调用的时候都会创建一个栈帧**，用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程就对应着一个栈帧在虚拟机中从入栈到出栈的过程。

在Java虚拟机规范中，对这个区域规定了两种异常情况:

（1）如果线程请求的栈深度太深，超出了虚拟机所允许的深度，就会出现StackOverFlowError（比如无限递归。因为每一层栈帧都占用一定空间，而 Xss 规定了栈的最大空间，超出这个值就会报错）

（2）虚拟机栈可以动态扩展，如果扩展到无法申请足够的内存空间，会出现OOM

4.1 Java栈之局部变量表：包含参数和局部变量

局部变量表存放了基本数据类型、对象引用和returnAddress类型（指向一条字节码指令的地址）。其中64位长度的long和double类型的数据会占用2个局部变量空间（slot），其余数据类型只占用1个。局部变量表所需的内存空间在编译期间完成分配。

例如，我写出下面这段代码：

```
1 package test03;
2
3 /**
4  * Created by smyhvae on 2015/8/15.
5  */
6 public class StackDemo {
7
8     //静态方法
9     public static int runStatic(int i, long l, float f, Object o,
byte b) {
10         return 0;
11     }
12
13     //实例方法
14     public int runInstance(char c, short s, boolean b) {
15         return 0;
16     }
17
18 }
```

上方代码中，静态方法有6个形参，实例方法有3个形参。其对应的局部变量表如下：

0	int	int i
1	long	long l
3	float	float f
4	reference	Object o
5	int	byte b

0	reference	this
1	int	char c
2	int	short s
3	int	boolean b

上方表格中，静态方法和实例方法对应的局部变量表基本类似。但有以下区别：实例方法的表中，第一个位置存放的是当前对象的引用。

4、2 Java栈之函数调用组成栈帧：

方法每次被调用的时候都会创建一个栈帧，例如下面这个方法：

```
public static int runStatic(int i,long l,float  f,Object o ,byte b){
        return runStatic(i,l,f,o,b);
}
```

当它每次被调用的时候，都会创建一个帧，方法调用结束后，帧出栈。如下图所示：

4.3 Java栈之操作数栈

Java没有寄存器，所有参数传递都是使用操作数栈

例如下面这段代码：

```
    public static int add(int a,int b){
```

```
        int c=0;
        c=a+b;
        return c;
    }
```

压栈的步骤如下：

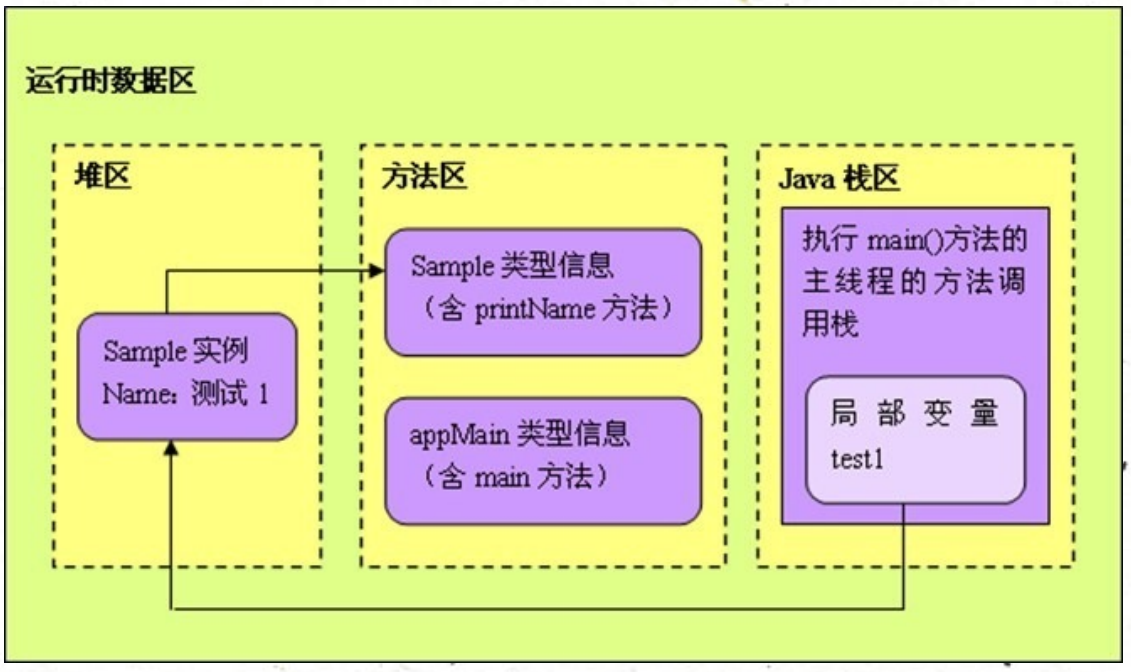
0: iconst_0 // 0压栈
1: istore_2 // 弹出int，存放于局部变量2
2: iload_0 // 把局部变量0压栈
3: iload_1 // 局部变量1压栈
4: iadd //弹出2个变量，求和，结果压栈
5: istore_2 //弹出结果，放于局部变量2
6: iload_2 //局部变量2压栈
7: ireturn //返回

如果计算100+98的值，那么操作数栈的变化如下图所示：

4.4 Java栈之栈上分配：

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

栈、堆、方法区交互：

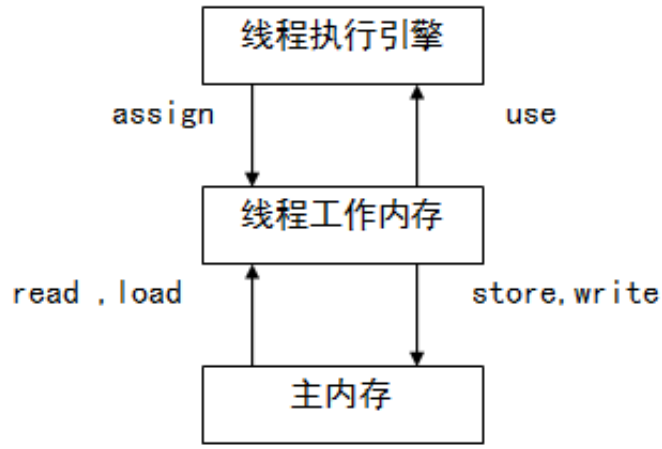



```
public  class  AppMain
 //运行时, jvm 把appmain的信息都放入方法区
{
public  static  void  main(String[] args)
//main 方法本身放入方法区。
{
Sample test1 = new  Sample( " 测试1 " );
 //test1是引用，所以放到栈区里， Sample是自定义对象应该放
到堆里面
Sample test2 = new  Sample( " 测试2 " );

test1.printName();
test2.printName();
}
```

```
public  class  Sample
 //运行时, jvm 把appmain的信息都放入方法区
{
private  name;
 //new Sample实例后， name 引用放入栈区里， name 对象
放入堆里

public  Sample(String name)
{
this .name = name;
}
//print方法本身放入 方法区里。
public  void  printName()
{
System.out.println(name); } }
```

三、内存模型：

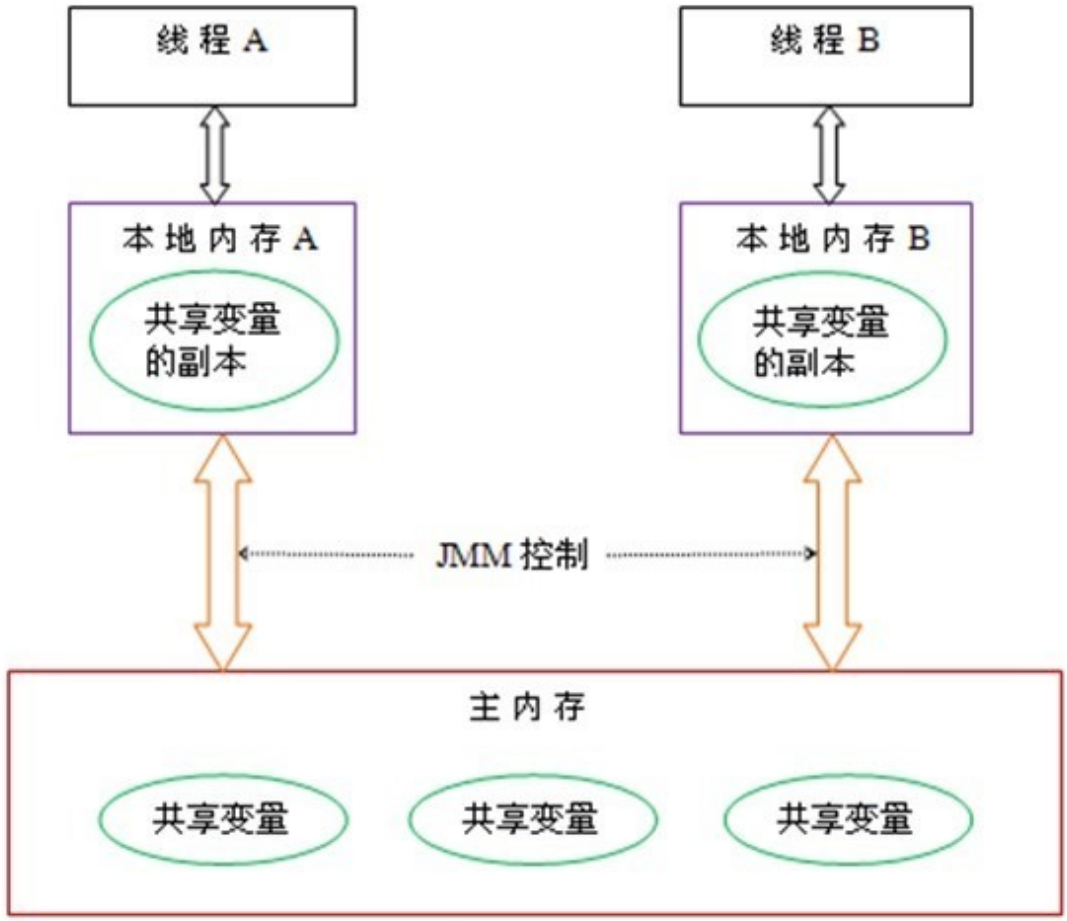
每一个线程有一个工作内存。工作内存和主存独立。工作内存存放主存中变量的值的拷贝。



当数据从主内存复制到工作存储时，必须出现两个动作：第一，由主内存执行的读（read）操作；第二，由工作内存执行的相应的load操作；当数据从工作内存拷贝到主内存时，也出现两个操作：第一个，由工作内存执行的存储（store）操作；第二，由主内存执行的相应的写（write）操作。

每一个操作都是原子的，即执行期间不会被中断

对于普通变量，一个线程中更新的值，不能马上反应在其他变量中。如果需要在其他线程中立即可见，需要使用volatile关键字作为标识。



1、可见性：

一个线程修改了变量，其他线程可以立即知道

保证可见性的方法：

volatile

synchronized （unlock之前，写变量值回主存）

final(一旦初始化完成，其他线程就可见)

2、有序性：

在本线程内，操作都是有序的

在线程外观察，操作都是无序的。（指令重排 或 主内存同步延时）

3、指令重排：

- 线程内串行语义
  - 写后读　　a = 1;b = a;　　写一个变量之后，再读这个位置。
  - 写后写　　a = 1;a = 2;　　写一个变量之后，再写这个变量。
  - 读后写　　a = b;b = 1;　　读一个变量之后，再写这个变量。
  - 以上语句不可重排
  - 编译器不考虑多线程间的语义
  - 可重排： a=1;b=2;

指令重排：破坏了线程间的有序性：

```
class OrderExample {
int a = 0;
boolean flag = false;

public void writer() {
    a = 1;
    flag = true;
}

public void reader() {
    if (flag) {
        int i =  a +1;
        ……
    }
}
}
```

线程A首先执行writer()方法
线程B线程接着执行reader()方法
线程B在int i=a+1 是不一定能看到a已经被赋值为1

因为在writer中，两句话顺序可能打乱

线程A	线程B
flag=true a=1	flag=true(此时a=0)

指令重排：保证有序性的方法：

```
class OrderExample {
int a = 0;
boolean flag = false;

public synchronized void writer() {
    a = 1;
    flag = true;
}

public synchronized void reader() {
    if (flag) {
        int i =  a +1;
        ……
    }
}
}
```

同步后，即使做了writer重排，因为互斥的缘故，
reader 线程看writer线程也是顺序执行的。

线程A	线程B
flag=true a=1	flag=true(此时a=1)

指令重排的基本原则：

程序顺序原则：一个线程内保证语义的串行性

volatile规则：volatile变量的写，先发生于读

锁规则：解锁(unlock)必然发生在随后的加锁(lock)前

传递性：A先于B，B先于C 那么A必然先于C

线程的start方法先于它的每一个动作

线程的所有操作先于线程的终结（Thread.join()）

线程的中断（interrupt()）先于被中断线程的代码

对象的构造函数执行结束先于finalize()方法

四、解释运行和编译运行的概念：

解释运行：

解释执行以解释方式运行字节码

解释执行的意思是：读一句执行一句

编译运行（JIT）：

将字节码编译成机器码

直接执行机器码

运行时编译

编译后性能有数量级的提升

编译运行的性能优于解释运行。