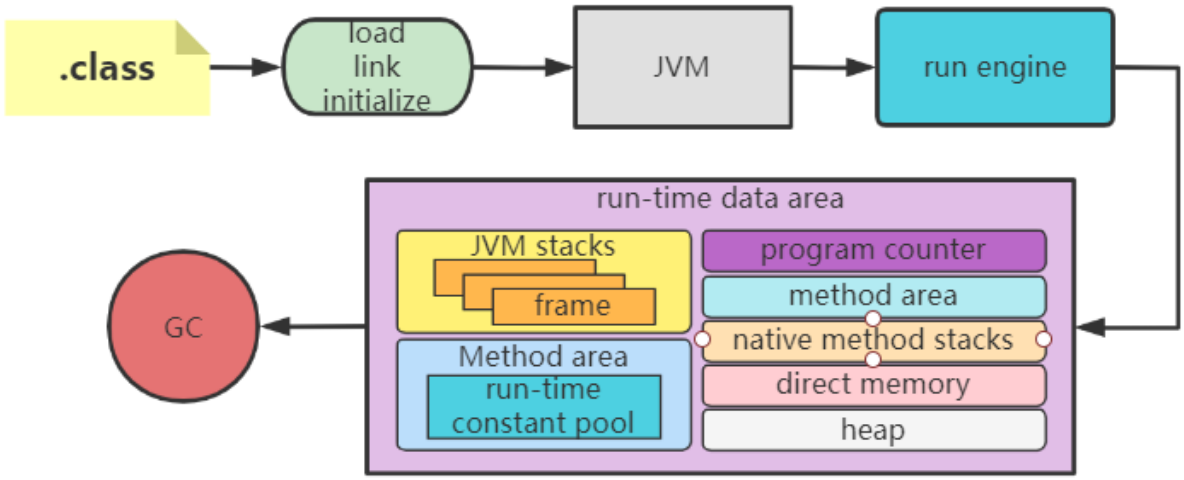
JVM Runtime Data Area And JVM Instructions

Class 的生命周期



Class 被jvm load 完后 会进入jvm的运行时引擎

run-time constant pool：运行的时候将常量内容仍到这个区域里面

ProgramCounter：程序计数器存放下一条指令位置的内存区域，虚拟机的运行

虚拟机的运行，类似于这样的循环：

```
while( not end ) {
        取PC中的位置，找到对应位置的指令;
        执行该指令;
        PC ++;
}
```

Heap：GC里面会讲

native method stacks： 本地方法（c/c++—→java调用了JNI） ，调用了内部 c 和 c++ 写的方法时的栈，一般不管这个，也没办法去调优

JVM stacks：java内部JVM管理的栈，java 运行的时候，每一个线程都有一个栈，装的是栈帧 frame

Direct Memory：java 1.4 之后增加的，JVM可以直接访问的内核空间的内存（OS管理内存）。直接内存区域，不归JVM管理，归操作系统管。一般情况下所有的内存一般都是JVM直接管理，为了增加IO的效率1.4之后增加了直接内存的概念，也就是在jave虚拟机实际是可以访问操作系统里面的内存的，提高效率。

> 理解：原来jvm在用到内核空间的内存的时候，首先要拷贝一份到jvm空间，有一个内存拷贝的过程，效率比较低。所以在1.4之后增加了NIO，直接可以操作内存，省去了拷贝的这么一个过程。也叫0拷贝—zero copy

Method area：装的是各种各样的class和常量池的内容

线程共享区域

每一个线程都有自己的 Program counter——目的是线程切换

每一个线程都有自己的 JVM stacks 装的是一个个 栈帧 frame，每个方法都有自己独立的栈帧

每一个线程都有自己的native method stack

共享 Heap （堆）

共享 Method Area

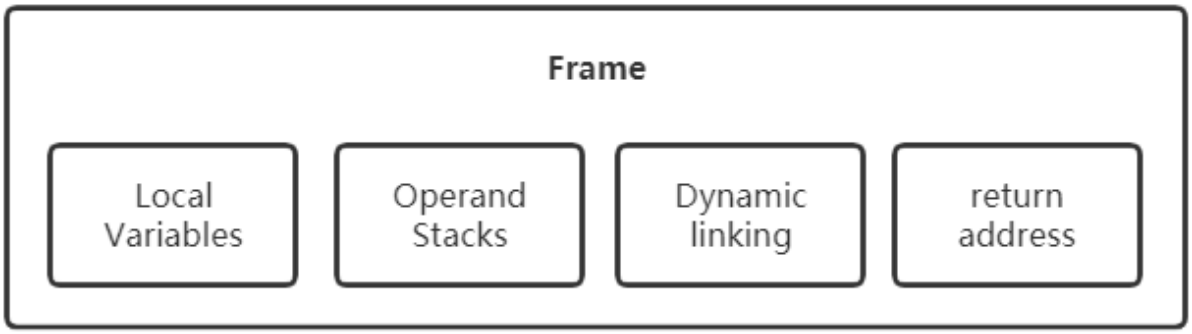
栈帧 Frame —每个方法对应一个栈帧

> A frame is used to store data and partial result，as well as to perform dynamic linking,return values for method,and dispatch exceptions
> 翻译：帧用于存储数据和部分结果，以及执行动态链接、方法返回值和分派异常

作用：存储数据，执行动态链接，返回方法数据，调度异常

每个线程都有自己的 JVM Stacks，每一个jvm stack 存放的是好多个栈帧，每一个栈帧里面都有自己的操作数栈

每个方法的栈帧都有如下几个组成部分



Local Variables： 局部变量

Operand Stacks 操作数栈， 计算的时候压栈出栈的区域

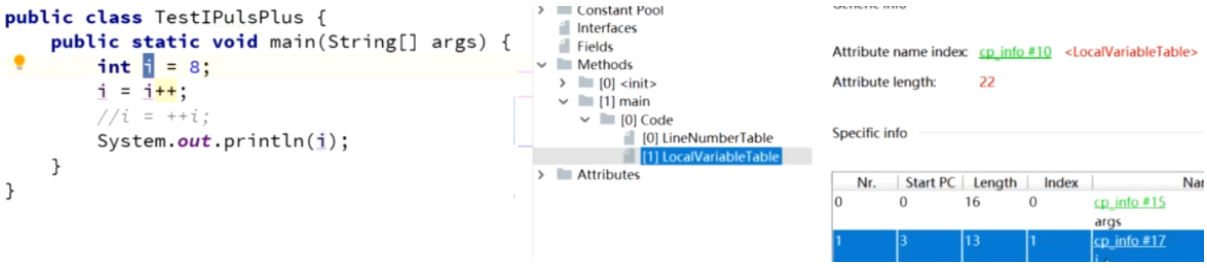
Dynamic linking 动态链接

return address 返回地址

Local variables Table

指的是当前方法用到的局部变量，栈帧弹出就没了

如下代码说明这个局部变量表有两个局部变量 args 和 i 名字信息记录在Constant pool里面



Dynamic Linking

作用：

从constant pool 找到符号链接，看有没有解析成直接引用，如果没有解析则动态解析，如果已经解析，则直接拿来用

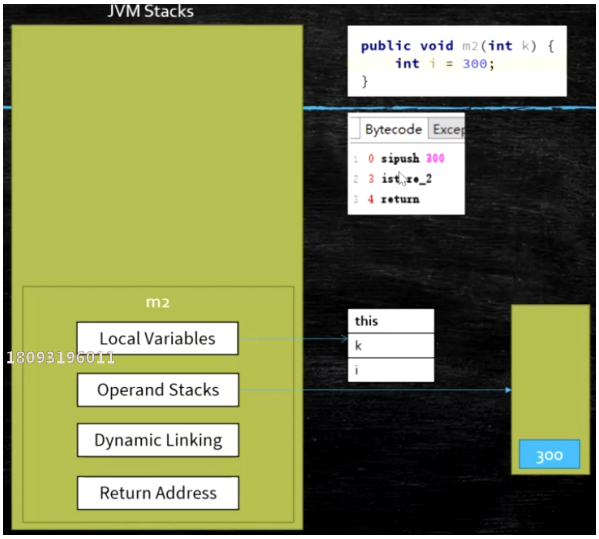
A 方法 调用了B 方法，而B 方法要去常量池中找，这个过程就叫Dynamic Linking

参考：https://blog.csdn.net/qq_41813060/article/details/88379473

return address

a()→b();

方法 a 调用了 b ，b 方法执行结束之后，b 方法的返回值放在什么地方，以及b方法结束之后应该回到那个地方继续执行　的这么一个过程



sipush　s==short

bipush　b = short

局部变量表如果是非静态方法，则有this这个局部变量，在第0个位置

Operand Stack 标识自己运算的一个 栈

局部变量表大小为4

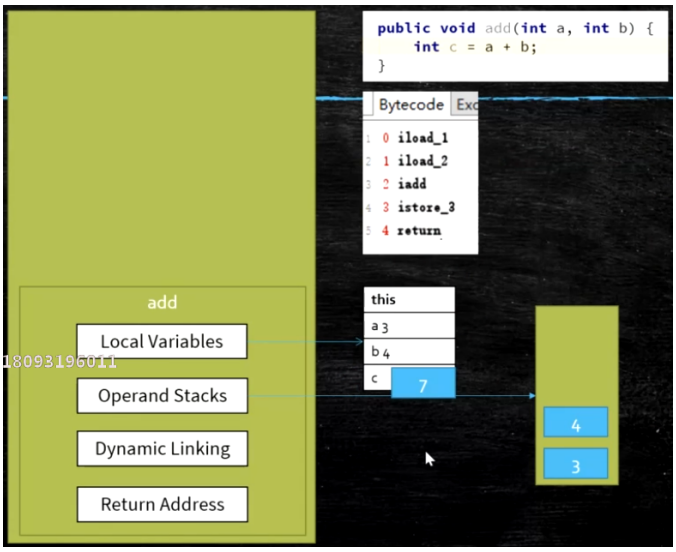
0：this

1：a

2：b

3：c

iload_1 压栈a　3 压栈

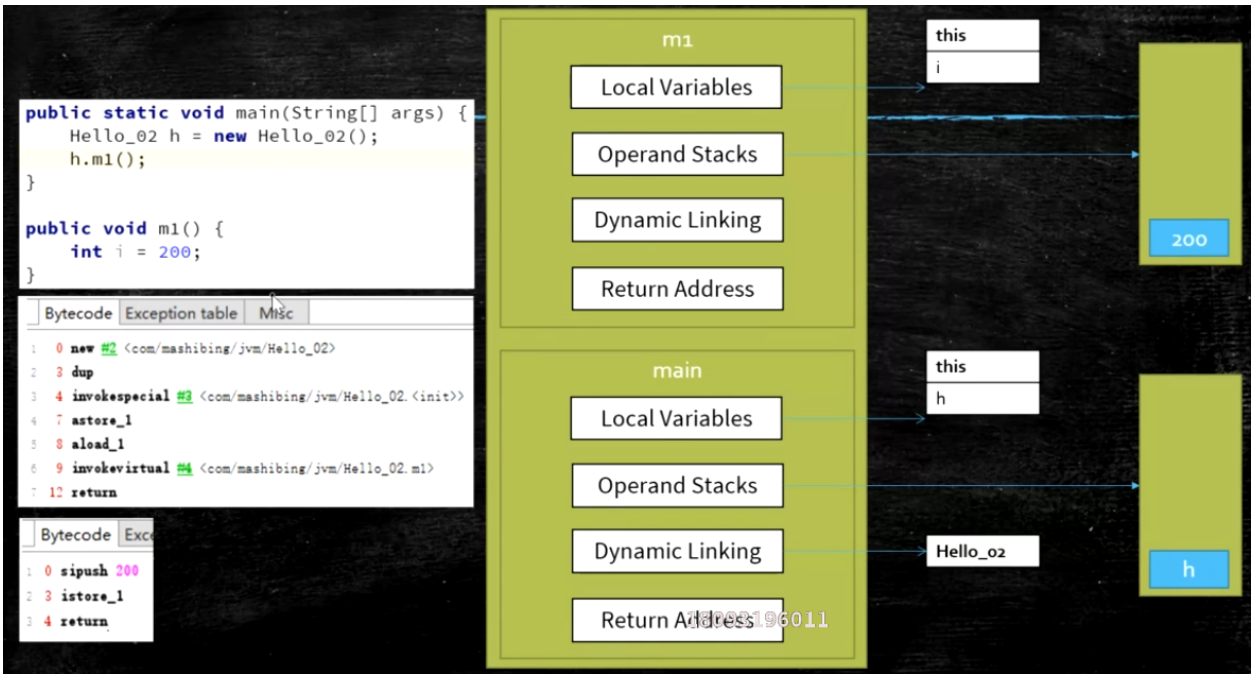


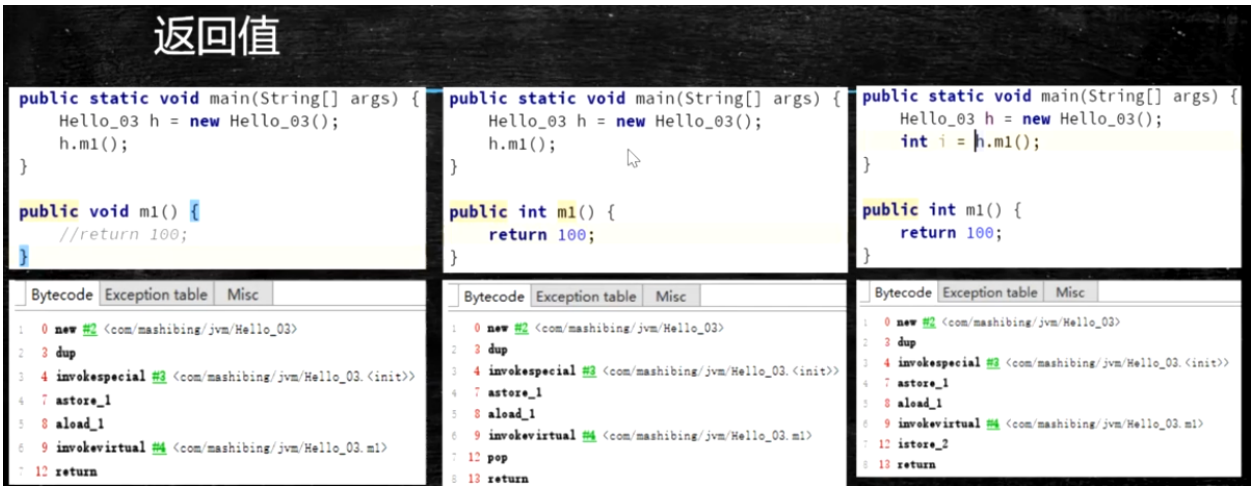
iload_2 压栈b　4压栈

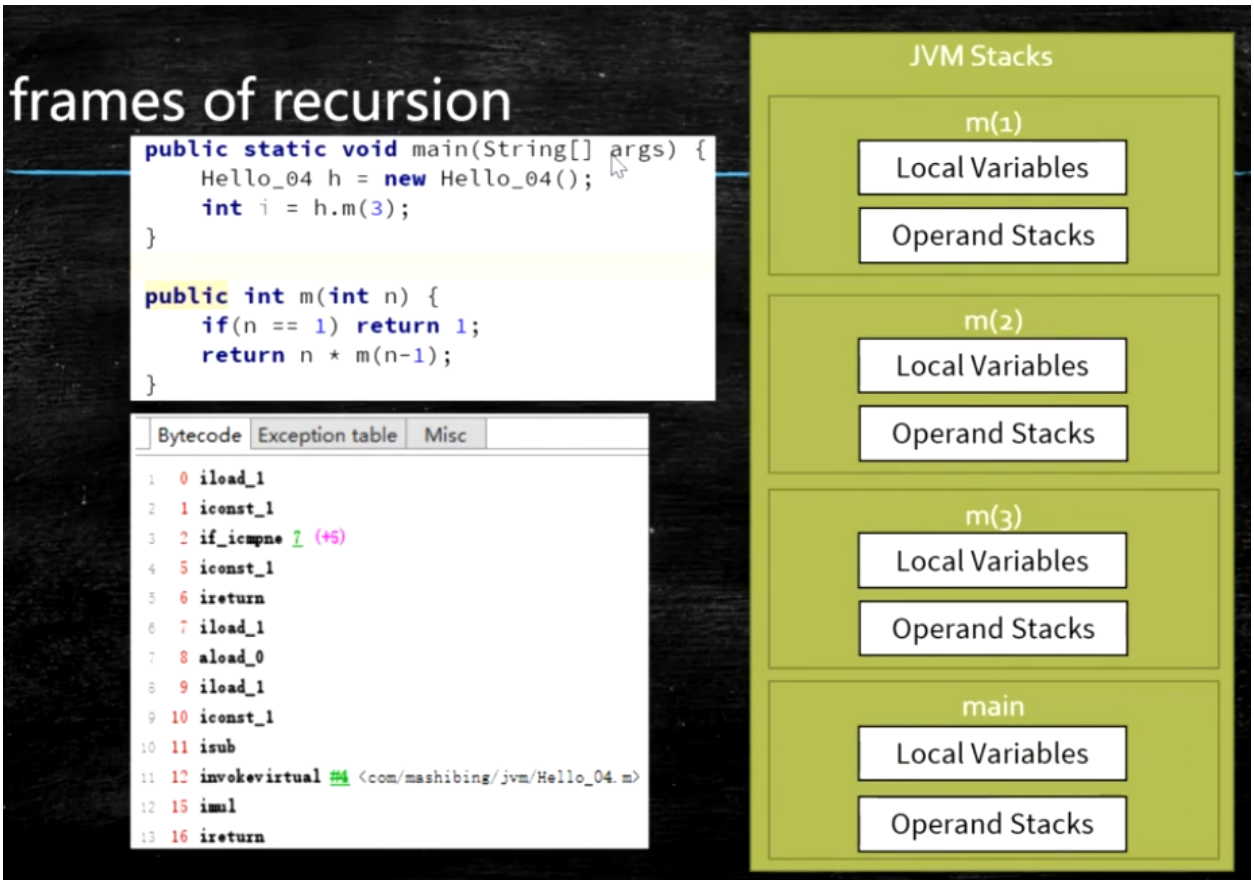
iadd 3 和4 出栈 相加算完之后压栈

istore_3 弹出来

return 返回







执行代码的时候 用到的变量 和常量压栈操作，赋值操作 表示出栈赋值

Method area

PermSpace 和 Meta Space 是不同版本对Method area的实现

1、Perm Space 1.8 之前　永久区域

字符串常量位于PermSpace
FGC不会清理
大小启动的时候指定，不能变

2、Meta Space 1.8以及以后　元数据区

字符串常量位于堆
会触发FGC清理
不设定的话，最大就是物理内存

常用指令

<clinit> class 静态语句块执行执行

<init>　构造方法

_store　出栈　　从栈里弹出来，然后本地变量表存储栈里面的东西

_load　压栈　　本地变量表的常量 加载进栈

pop 弹出

mul 乘法

sub 减法

add 加法

div 除法

invode

invokeStatic：调用静态方法用到的指令

invokeVirtual：调用普通方法用到的指令

自带多态，栈里面压得是哪个就是哪个

invokeSpecial：调用可以直接定位，不需要多态的方法，即：private 和 构造方法

invokeInterface：调用接口的方法

invokeDynamic：（最难）

lambda 表达式、或反射或者其他动态语言 会动态产生自己对应的Class，会用到该指令

```
for(;;){
    I aa = C :: c ;  // 这样会创建很多内部类，1.8 之前 经常OOM  1.8之后 回收不及时会OOM
}
```