课程主题

虚拟机栈、程序计数器、方法执行和方法调用&垃圾回收

课程目标

3. 掌握JVM中方法调用指令有哪些?
4. 掌握JVM中静态绑定是怎么回事?
5. 掌握JVM中动态绑定是怎么回事?
6. 掌握JVM中重载是如何确定调用哪个方法的?
7. 掌握垃圾回收的判断算法
8. 掌握垃圾对象是如何回收，也就是回收算法有哪些?
9. 掌握GC日志如何查看?
10. 搞清楚内存分配担保在Client模式是如何实现的?
11. 搞清楚内存分配担保在Server模式是如何实现的?

课程回顾

1. 搞清楚Java堆中存储什么数据?
    1. 对象
    2. 数组
    3. 静态变量
    4. 字符串常量池
2. 搞清楚Java堆的内存划分具体是如何操作的?
    1. JVM启动的时候，通过一些参数来将堆划分为年轻代、年老代
    2. 年轻代又划分为伊甸园区和两个同等大小的幸存区
3. 搞清楚Java堆中内存分配的原则是什么?
    1. 优先在伊甸园区进行对象分配
    2. 大对象会分配到老年代
    3. 当对象的年龄超过一定的阈值，也会进入到老年代
4. 搞清楚Java堆中内存分配的方式和多线程内存分配安全问题?
    1. 内存分配方式：指针碰撞、空闲列表
    2. 内存安全保证：CAS+TLAB
5. 搞清楚Java对象在堆中是存储成什么布局?
    1. 对象头：对象年龄、类型指针等
    2. 实例数据：对象的成员变量
    3. 对齐填充
6. 搞清楚Java堆中的对象是如何被访问的?
    1. 句柄访问
    2. 直接指针访问
7. 搞清楚Java堆中的数组是如何被访问的?

8. 可以通过jmap命令查看堆中的信息?
9. 搞清楚栈帧是什么?
    1. 就是为了保证方法执行和方法调用时提供的一个数据结构。
10. 搞清楚栈帧是什么时候创建的?
    1. 当一个方法被调用的时候，就会创建对应的栈帧
11. 搞清楚栈帧中都包含哪些内容?
    1. 局部变量表
    2. 操作数栈
    3. 方法返回地址
    4. 动态链接
12. 搞清楚局部变量表、操作数栈都是如何存储数据的?
    1. 局部变量表是通过数组去存储，每个数组的元素是一个变量槽，每个变量槽，可以存储32位的数据。
13. 了解本地方法栈是什么?
14. 了解为什么要划分一块本地方法栈的内存?
15. 了解本地方法和Java方法一起是如何工作的?
16. 了解程序计数器的作用以及为什么要有程序计数器?
17. 了解常用的字节码指令集
    1. const系列
    2. load系列
    3. store系列
    4. ldc系列
18. 可以分析通过javap命令显示的字节码指令信息，最终理解方法是如何执行
    1. 学会自己画图去实现

课程内容

一、程序计数器

作用

程序计数寄存器（Program Counter Register），也叫PC寄存器，也叫程序计数器。

是一块较小的内存空间，它可以看作是当前线程所执行的字节码指令的行号指示器。

> 字节码解释器的工作就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。

> 分支，循环，跳转，异常处理，线程回复等都需要依赖这个计数器来完成。

为什么使用程序计数器?

> 由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（针对多核处理器来说是一个内核）都只会执行一条线程中的指令。

因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

存储的数据

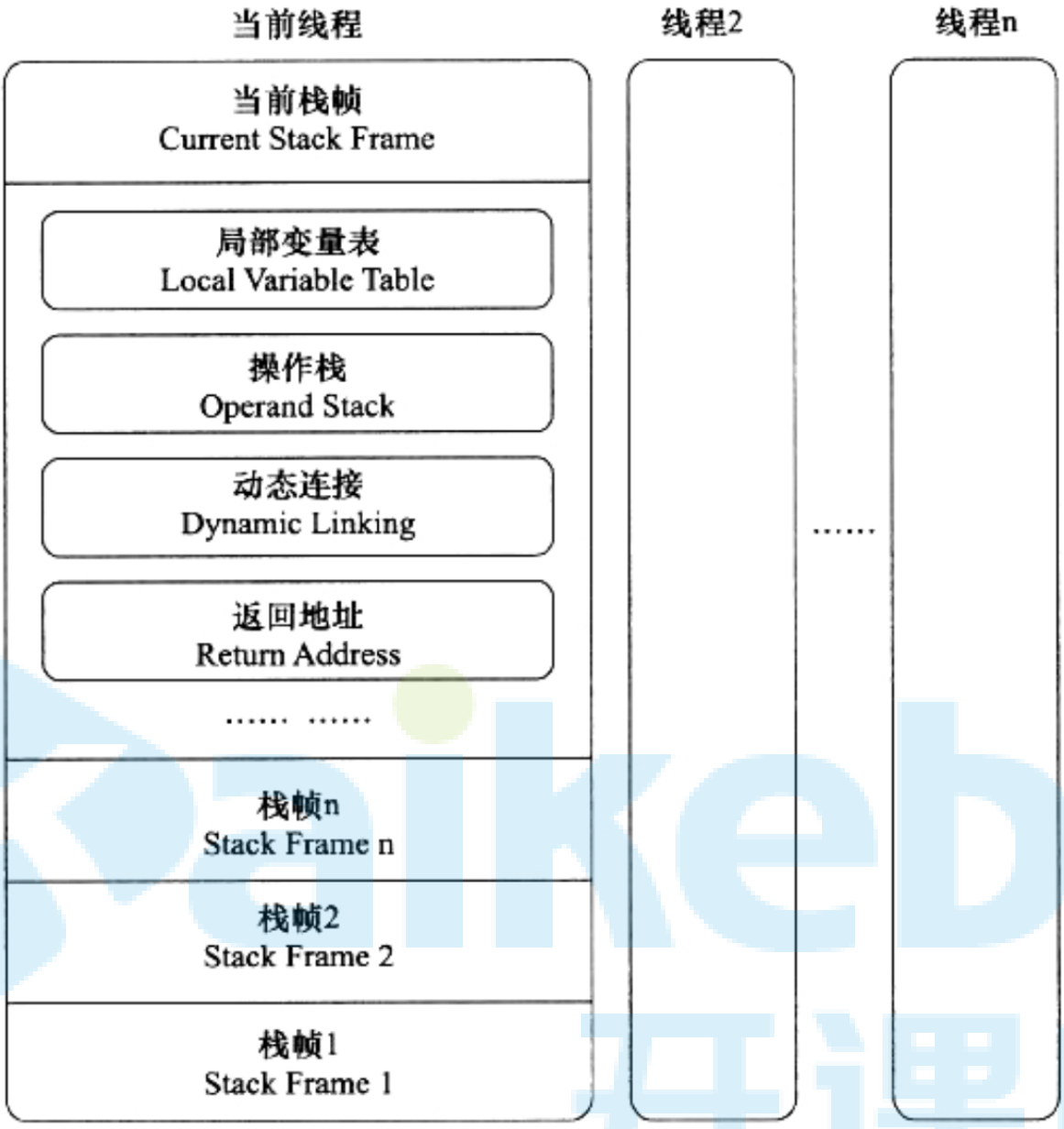
如果一个线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是一个Native方法，这个计数器的值则为空。

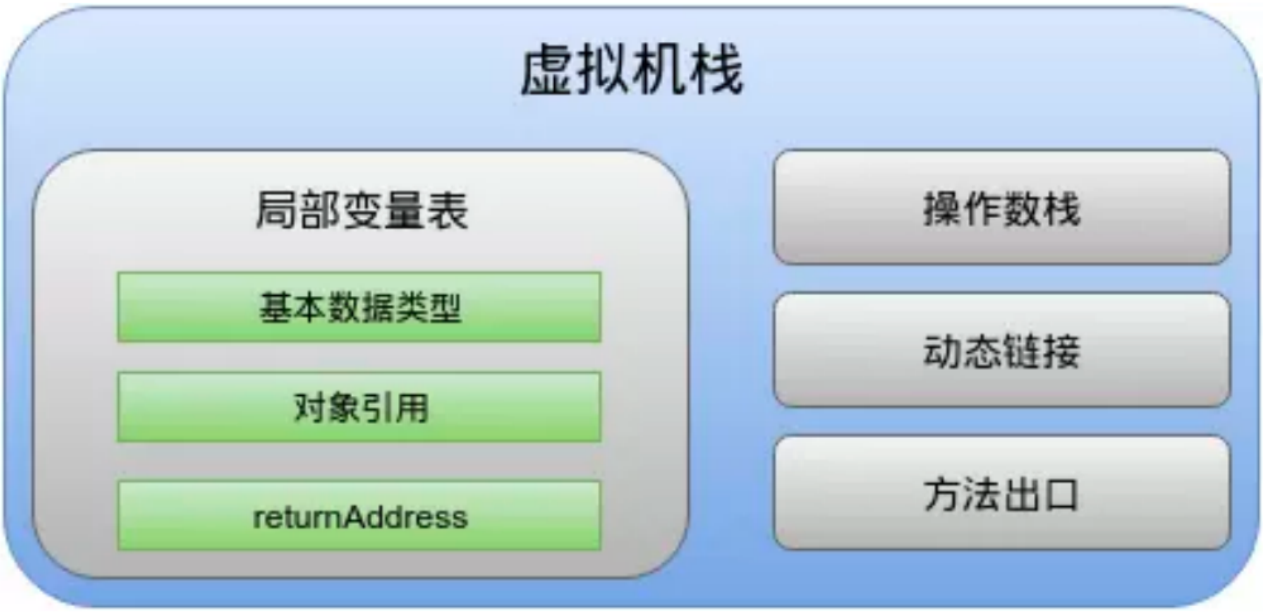
异常

此内存区域是唯一一个在Java的虚拟机规范中没有规定任何OutOfMemoryError异常情况的区域。

二、Java虚拟机栈

虚拟机栈也是线程私有，而且生命周期与线程相同，每个Java方法在执行的时候都会创建一个栈帧（Stack Frame）。


当前线程
线程2
线程n
当前栈帧
Current Stack Frame
局部变量表
Local Variable Table
操作栈
Operand Stack
动态连接
Dynamic Linking
返回地址
Return Address
…… ……
栈帧n
Stack Frame n
栈帧2
Stack Frame 2
栈帧1
Stack Frame 1
……



> 栈内存为线程私有的空间，每个线程都会创建私有的栈内存。
>
> 栈空间内存设置过大，创建线程数量较多时会出现栈内存溢出StackOverflowError。
>
> 同时，栈内存也决定方法调用的深度，栈内存过小则会导致方法调用的深度较小，如递归调用的次数较少。

栈帧

栈帧是什么

栈帧(Stack Frame)是用于支持虚拟机进行**方法执行**的数据结构。

> 栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用至执行完成的过程，都对应着一个栈帧在虚拟机栈里从入栈到出栈的过程。

当前栈帧

一个线程中方法的调用链可能会很长，所以会有很多栈帧。只有位于JVM虚拟机栈栈顶的元素才是有效的，即称为**当前栈帧**，与这个栈帧相关连的方法称为**当前方法**，定义这个方法的类叫做**当前类**。

执行引擎运行的所有**字节码指令**都只针对**当前栈帧**进行操作。如果当前方法调用了其他方法，或者当前方法执行结束，那这个方法的栈帧就不再是当前栈帧了。

什么时候创建栈帧

调用新的方法时，新的栈帧也会随之创建。并且随着程序控制权转移到新方法，新的栈帧成为了当前栈帧。方法返回之际，原栈帧会返回方法的执行结果给之前的栈帧(返回给方法调用者)，随后虚拟机将会丢弃此栈帧。

局部变量表

存储内容

局部变量表(Local Variable Table)是一组变量值存储空间，用于存放**方法参数**和**方法内定义的局部变量**。

一个局部变量可以保存一个类型为**boolean、byte、char、short、int、float、reference和returnAddress类型**的数据。**reference**类型表示对一个对象实例的引用。。

局部变量：

- this引用（实例对象都需要维护的一个变量，而且在局部变量表中始终处于第一个位置，也就是下标为0的位置）
- 方法参数
- 方法内声明的变量

存储容量

局部变量表的容量以**变量槽(Variable Slot)**为最小单位，Java虚拟机规范并没有定义一个槽所应该占用内存空间的大小，但是规定了一个槽应该可以存放一个32位以内的数据类型。

> 在Java程序编译为Class文件时，就在方法的Code属性中的max_locals数据项中确定了该方法所需分配的局部变量表的最大容量。(最大Slot数量)

double\long这种8字节类型的数据，都需要两个slot来存储。

其他

虚拟机通过索引定位的方法查找相应的局部变量，索引的范围是从**0~局部变量表最大容量**。如果Slot是32位的，则遇到一个64位数据类型的变量(如long或double型)时，会连续使用两个连续的Slot来存储。

操作数栈

作用

操作数栈(Operand Stack)也常称为操作栈，它是一个**后入先出栈(LIFO)**。

JVM的解释引擎是基于栈（操作数栈）的方式去执行的。（另外还有一种是基于寄存器的方式）

当一个方法刚刚开始执行时，其操作数栈是空的，随着方法执行和字节码指令的执行，会从**局部变量表**或**对象实例的字段**中复制常量或变量写入到操作数栈，再随着计算的进行将栈中元素出栈到局部变量表或者返回给方法调用者，也就是出栈/入栈操作。一个完整的方法执行期间往往包含多个这样出栈/入栈的过程。

存储内容

操作数栈的每一个元素可以是任意Java数据类型，32位的数据类型占一个栈容量，64位的数据类型占2个栈容量。

存储容量

同局部变量表一样，操作数栈的最大深度也在编译的时候写入到方法的Code属性的**max_stacks**数据项中。且在方法执行的任意时刻，操作数栈的深度都不会超过**max_stacks**中设置的最大值。

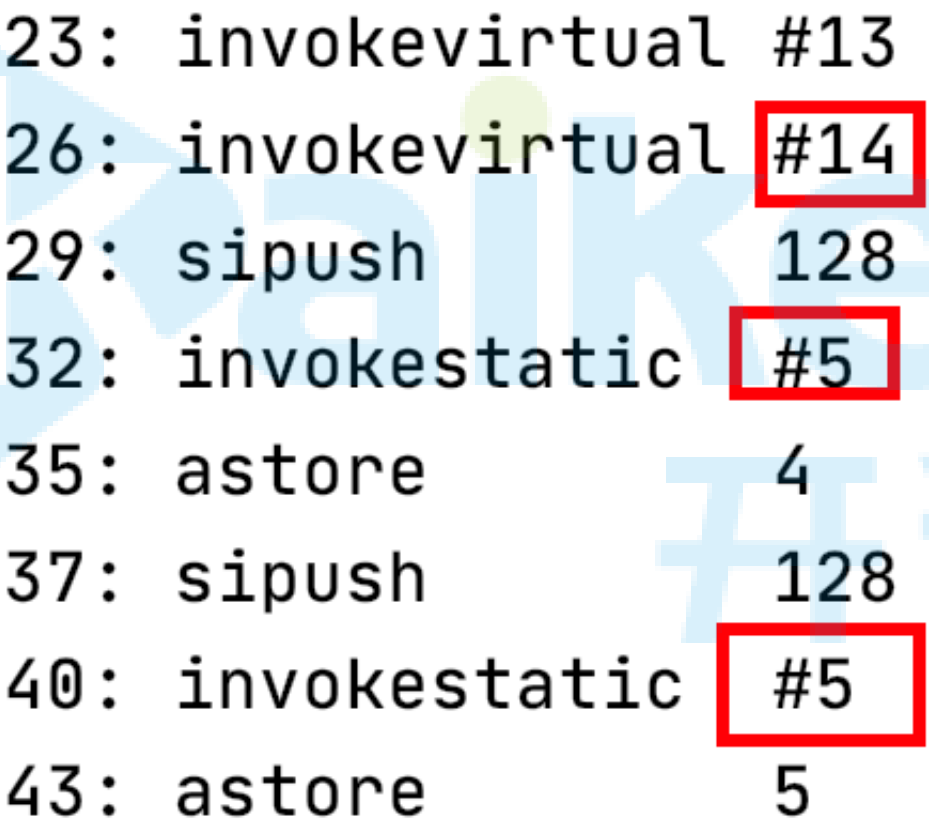
结论

一个线程的执行过程中，需要进行两个栈的入栈出栈操作，一个是JVM栈（栈帧的出栈和入栈），一个是操作数栈（参与计算的值进行出栈和入栈）

动态链接（Dynamic Linking）

在一个class文件中，一个方法要调用其他方法，需要将这些方法的符号引用转化为其在内存地址中的**直接引用**，而符号引用存在于方法区中的运行时常量池。

- 符号引用：



- 直接引用：

就是对应方法的内存地址

Java虚拟机栈中，每个栈帧都包含一个指向运行时常量池中该栈所属方法的符号引用，持有这个引用的目的是为了支持方法调用过程中的**动态连接(Dynamic Linking)**。

这些**符号引用**一部分会在**类加载阶段**或者**第一次使用**时就**直接转化为直接引用**，这类转化称为**静态解析**。另一部分将在每次运行期间转化为直接引用，这类转化称为动态连接。

方法返回

当一个方法开始执行时，可能有两种方式退出该方法：

- **正常完成出口**
- **异常完成出口**

正常完成出口是指方法正常完成并退出，没有抛出任何异常(包括Java虚拟机异常以及执行时通过throw语句显示抛出的异常)。如果当前方法正常完成，则根据当前方法返回的字节码指令，这时有可能会有返回值传递给方法调用者(调用它的方法)，或者无返回值。具体是否有返回值以及返回值的数据类型将根据该方法返回的字节码指令确定。

异常完成出口是指方法执行过程中遇到异常，并且这个异常在方法体内部没有得到处理，导致方法退出。

> 无论是Java虚拟机抛出的异常还是代码中使用athrow指令产生的异常，只要在本方法的异常表中没有搜索到相应的异常处理器，就会导致方法退出。

无论方法采用何种方式退出，在方法退出后都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在当前栈帧中保存一些信息，用来帮他恢复它的上层方法执行状态。

> A() ---> B() ，当B方法执行完返回时，发生以下操作：
>
> 方法退出过程实际上就等同于把当前栈帧出栈，因此退出可以执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值(如果有的话)压入调用者的操作数栈中，调整PC计数器的值以指向方法调用指令后的下一条指令。

一般来说，方法正常退出时，调用者的PC计数值可以作为返回地址，栈帧中可能保存此计数值。而方法异常退出时，返回地址是通过异常处理器表确定的，栈帧中一般不会保存此部分信息。

栈异常

Java虚拟机规范中，对该区域规定了这两种异常情况：

1. 如果线程请求的栈深度大于虚拟机所允许的深度，将会抛出`StackOverflowError`异常（-Xss）；
2. 虚拟机栈可以动态拓展，当扩展时无法申请到足够的内存，就会抛出`OutOfMemoryError`异常。

```
package com.kkb.test.memory;

public class StackErrorMock {
    private static int index = 1;
```

```
    public void call(){
        index++;
        call();
    }

    public static void main(String[] args) {
        StackErrorMock mock = new StackErrorMock();
        try {
            mock.call();
        }catch (Throwable e){
            System.out.println("Stack deep : "+index);
            e.printStackTrace();
        }
    }
}
```

三、本地方法栈

什么是本地方法

本地方法栈和虚拟机栈相似，区别就是虚拟机栈为虚拟机执行**Java服务（字节码服务）**，而本地方法栈为虚拟机使用到的**Native方法（比如C++方法）服务**。

> 简单地讲，一个Native Method就是一个java调用非java代码的接口。**一个Native Method是这样一个java的方法：该方法的实现由非java语言实现，比如C。**

在定义一个native method时，并不提供实现体（有些像定义一个java interface），因为其实现体是由非java语言在外面实现的。下面给了一个示例：

```
public class IHaveNatives
   {
     native public void Native1( int x ) ;
     native static public long Native2() ;
     native synchronized private float Native3( Object o ) ;
     native void Native4( int[] ary ) throws Exception ;
 }
```

为什么要使用本地方法

java使用起来非常方便，然而有些层次的任务用java实现起来不容易，或者我们对程序的效率很在意时，问题就来了。

本地方法非常有用，因为它有效地扩充了jvm。

> 事实上，我们所写的java代码已经用到了本地方法，在sun的java的并发（多线程）的机制实现中，许多与操作系统的接触点都用到了本地方法，这使得java程序能够超越java运行时的界限。有了本地方法，java程序可以做任何应用层次的任务。

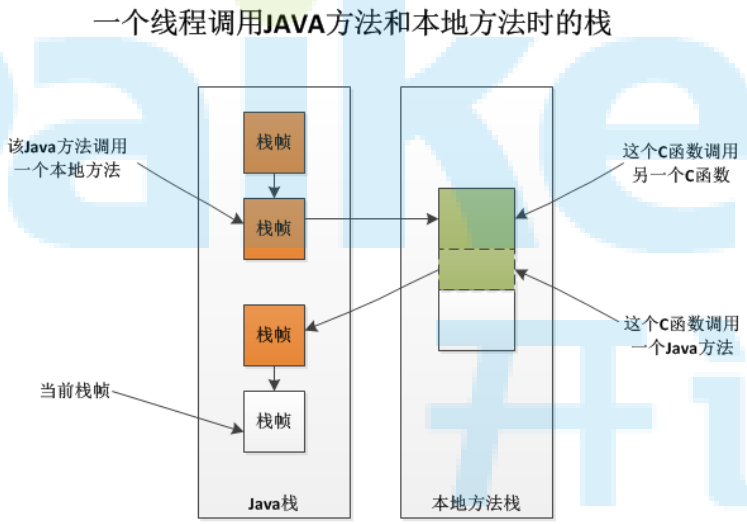
有时java应用需要与java外面的环境交互。这是本地方法存在的主要原因，你可以想想java需要与一些底层系统如操作系统或某些硬件交换信息时的情况。

> 本地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解java应用之外的繁琐的细节。

本地方法栈的使用流程

下图描绘了这样一个情景，就是当一个线程调用一个本地方法时，本地方法又回调虚拟机中的另一个Java方法。

这幅图展示了JAVA虚拟机内部线程运行的全景图。一个线程可能在整个生命周期中都执行Java方法，操作它的Java栈；或者它可能毫无障碍地在Java栈和本地方法栈之间跳转。



该线程首先调用了两个Java方法，而第二个Java方法又调用了一个本地方法，这样导致虚拟机使用了一个本地方法栈。假设这是一个C语言栈，其间有两个C函数，第一个C函数被第二个Java方法当做本地方法调用，而这个C函数又调用了第二个C函数。之后第二个C函数又通过本地方法接口回调了一个Java方法（第三个Java方法），最终这个Java方法又调用了一个Java方法（它成为图中的当前方法）。

四、方法执行

字节码指令集(字典)

概述

Java虚拟机的指令由一个字节长度的、代表着某种特定操作含义的数字（称为操作码，Opcode）以及跟随其后的零至多个代表此操作所需参数（称为操作数，Operands）而构成。

Opcode+操作数

- iconst_0　操作码
- bipush 10　操作码+操作数

比如：

字节码	助记符	指令含义
0x00	nop	什么都不做
0x01	aconst_null	将 null 推送至栈顶
0x02	iconst_m1	将 int 型 -1 推送至栈顶
0x03	iconst_0	将 int 型 0 推送至栈顶
0x04	iconst_1	将 int 型 1 推送至栈顶
0x05	iconst_2	将 int 型 2 推送至栈顶
0x06	iconst_3	将 int 型 3 推送至栈顶
0x07	iconst_4	将 int 型 4 推送至栈顶
0x08	iconst_5	将 int 型 5 推送至栈顶
0x09	lconst_0	将 long 型 0 推送至栈顶
0x0a	lconst_1	将 long 型 1 推送至栈顶
0x0b	fconst_0	将 float 型 0 推送至栈顶
0x0c	fconst_1	将 float 型 1 推送至栈顶
0x0d	fconst_2	将 float 型 2 推送至栈顶
0x0e	dconst_0	将 double 型 0 推送至栈顶
0x0f	dconst_1	将 double 型 1 推送至栈顶
0x10	bipush	将单字节的常量值（Byte.MIN_VALUE ～ Byte.MAX_VALUE，即 -128～127）推送至栈顶
0x11	sipush	将短整型的常量值（Short.MIN_VALUE ～ Short.MAX_VALUE，即 -32768～32767）推送至栈顶
0x12	ldc	将 int、float 或 String 型常量值从常量池中推送至栈顶

0x13	ldc_w	将 int、float 或 String 型常量值从常量池中推送至栈顶（宽索引）
0x14	ldc2_w	将 long 或 double 型常量值从常量池中推送至栈顶（宽索引）
0x15	iload	将指定的 int 型局部变量推送至栈顶
0x16	lload	将指定的 long 型局部变量推送至栈顶
0x17	fload	将指定的 float 型局部变量推送至栈顶
0x18	dload	将指定的 double 型局部变量推送至栈顶
0x19	aload	将指定的 引用 型局部变量推送至栈顶
0x1a	iload_0	将第一个 int 型局部变量推送至栈顶
0x1b	iload_1	将第二个 int 型局部变量推送至栈顶
0x1c	iload_2	将第三个 int 型局部变量推送至栈顶
0x1d	iload_3	将第四个 int 型局部变量推送至栈顶
0x1e	lload_0	将第一个 long 型局部变量推送至栈顶
0x1f	lload_1	将第二个 long 型局部变量推送至栈顶
0x20	lload_2	将第三个 long 型局部变量推送至栈顶
0x21	lload_3	将第四个 long 型局部变量推送至栈顶
0x22	fload_0	将第一个 float 型局部变量推送至栈顶
0x23	fload_1	将第二个 float 型局部变量推送至栈顶
0x24	fload_2	将第三个 float 型局部变量推送至栈顶
0x25	fload_3	将第四个 float 型局部变量推送至栈顶
0x26	dload_0	将第一个 double 型局部变量推送至栈顶
0x27	dload_1	将第二个 double 型局部变量推送至栈顶
0x28	dload_2	将第三个 double 型局部变量推送至栈顶
0x29	dload_3	将第四个 double 型局部变量推送至栈顶
0x2a	aload_0	将第一个 引用 型局部变量推送至栈顶
0x2b	aload_1	将第二个 引用 型局部变量推送至栈顶
0x2c	aload_2	将第三个 引用 型局部变量推送至栈顶
0x2d	aload_3	将第四个 引用 型局部变量推送至栈顶
0x2e	iaload	将 int 型数组指定索引的值推送至栈顶

0x2f	laload	将 long 型数组指定索引的值推送至栈顶
0x30	faload	将 float 型数组指定索引的值推送至栈顶
0x31	daload	将 double 型数组指定索引的值推送至栈顶
0x32	aaload	将 引用 型数组指定索引的值推送至栈顶
0x33	baload	将 boolean 或 byte 型数组指定索引的值推送至栈顶
0x34	caload	将 char 型数组指定索引的值推送至栈顶
0x35	saload	将 short 型数组指定索引的值推送至栈顶
0x36	istore	将栈顶 int 型数值存入指定局部变量
0x37	lstore	将栈顶 long 型数值存入指定局部变量
0x38	fstore	将栈顶 float 型数值存入指定局部变量
0x39	dstore	将栈顶 double 型数值存入指定局部变量
0x3a	astore	将栈顶 引用 型数值存入指定局部变量
0x3b	istore_0	将栈顶 int 型数值存入第一个局部变量
0x3c	istore_1	将栈顶 int 型数值存入第二个局部变量
0x3d	istore_2	将栈顶 int 型数值存入第三个局部变量
0x3e	istore_3	将栈顶 int 型数值存入第四个局部变量
0x3f	lstore_0	将栈顶 long 型数值存入第一个局部变量
0x40	lstore_1	将栈顶 long 型数值存入第二个局部变量
0x41	lstore_2	将栈顶 long 型数值存入第三个局部变量
0x42	lstore_3	将栈顶 long 型数值存入第四个局部变量
0x43	fstore_0	将栈顶 float 型数值存入第一个局部变量
0x44	fstore_1	将栈顶 float 型数值存入第二个局部变量
0x45	fstore_2	将栈顶 float 型数值存入第三个局部变量
0x46	fstore_3	将栈顶 float 型数值存入第四个局部变量
0x47	dstore_0	将栈顶 double 型数值存入第一个局部变量
0x48	dstore_1	将栈顶 double 型数值存入第二个局部变量
0x49	dstore_2	将栈顶 double 型数值存入第三个局部变量
0x4a	dstore_3	将栈顶 double 型数值存入第四个局部变量

0x4b	astore_0	将栈顶 引用 型数值存入第一个局部变量
0x4c	astore_1	将栈顶 引用 型数值存入第二个局部变量
0x4d	astore_2	将栈顶 引用 型数值存入第三个局部变量
0x4e	astore_3	将栈顶 引用 型数值存入第四个局部变量
0x4f	iastore	将栈顶 int 型数值存入指定数组的指定索引位置
0x50	lastore	将栈顶 long 型数值存入指定数组的指定索引位置
0x51	fastore	将栈顶 float 型数值存入指定数组的指定索引位置
0x52	dastore	将栈顶 double 型数值存入指定数组的指定索引位置
0x53	aastore	将栈顶 引用 型数值存入指定数组的指定索引位置
0x54	bastore	将栈顶 boolean 或 byte 型数值存入指定数组的指定索引位置
0x55	castore	将栈顶 char 型数值存入指定数组的指定索引位置
0x56	sastore	将栈顶 short 型数值存入指定数组的指定索引位置
0x57	pop	将栈顶数值弹出（数值不能是 long 或 double 类型的）
0x58	pop2	将栈顶的一个（对于 long 或 double 类型）或两个数值（对于非 long 或 double 的其他类型）弹出
0x59	dup	复制栈顶数值并将复制值压入栈顶
0x5a	dup_x1	复制栈顶数值并将两个复制值压入栈顶
0x5b	dup_x2	复制栈顶数值并将三个（或两个）复制值压入栈顶
0x5c	dup2	复制栈顶一个（对于 long 或 double 类型）或两个数值（对于非 long 或 double 的其他类型）并将复制值压入栈顶
0x5d	dup2_x1	dup_x1 指令的双倍版本
0x5e	dup2_x2	dup_x2 指令的双倍版本
0x5f	swap	将栈最顶端的两个数值互换（数值不能是 long 或 double 类型）
0x60	iadd	将栈顶两 int 型数值相加并将结果压入栈顶
0x61	ladd	将栈顶两 long 型数值相加并将结果压入栈顶
0x62	fadd	将栈顶两 float 型数值相加并将结果压入栈顶
0x63	dadd	将栈顶两 double 型数值相加并将结果压入栈顶
0x64	isub	将栈顶两 int 型数值相减并将结果压入栈顶
0x65	lsub	将栈顶两 long 型数值相减并将结果压入栈顶

0x66	fsub	将栈顶两 float 型数值相减并将结果压入栈顶
0x67	dsub	将栈顶两 double 型数值相减并将结果压入栈顶
0x68	imul	将栈顶两 int 型数值相乘并将结果压入栈顶
0x69	lmul	将栈顶两 long 型数值相乘并将结果压入栈顶
0x6a	fmul	将栈顶两 float 型数值相乘并将结果压入栈顶
0x6b	dmul	将栈顶两 double 型数值相乘并将结果压入栈顶
0x6c	idiv	将栈顶两 int 型数值相除并将结果压入栈顶
0x6d	ldiv	将栈顶两 long 型数值相除并将结果压入栈顶
0x6e	fdiv	将栈顶两 float 型数值相除并将结果压入栈顶
0x6f	ddiv	将栈顶两 double 型数值相除并将结果压入栈顶
0x70	irem	将栈顶两 int 型数值作取模运算并将结果压入栈顶
0x71	lrem	将栈顶两 long 型数值作取模运算并将结果压入栈顶
0x72	frem	将栈顶两 float 型数值作取模运算并将结果压入栈顶
0x73	drem	将栈顶两 double 型数值作取模运算并将结果压入栈顶
0x74	ineg	将栈顶两 int 型数值取负并将结果压入栈顶
0x75	lneg	将栈顶两 long 型数值取负并将结果压入栈顶
0x76	fneg	将栈顶两 float 型数值取负并将结果压入栈顶
0x77	dneg	将栈顶两 double 型数值取负并将结果压入栈顶
0x78	ishl	将 int 型数值左移指定位数并将结果压入栈顶
0x79	lshl	将 long 型数值左移指定位数并将结果压入栈顶
0x7a	ishr	将 int 型数值右（带符号）移指定位数并将结果压入栈顶
0x7b	lshr	将 long 型数值右（带符号）移指定位数并将结果压入栈顶
0x7c	iushr	将 int 型数值右（无符号）移指定位数并将结果压入栈顶
0x7d	lushr	将 long 型数值右（无符号）移指定位数并将结果压入栈顶
0x7e	iand	将栈顶两 int 型数值作“按位与”并将结果压入栈顶
0x7f	land	将栈顶两 long 型数值作“按位与”并将结果压入栈顶
0x80	ior	将栈顶两 int 型数值作“按位或”并将结果压入栈顶
0x81	lor	将栈顶两 long 型数值作“按位或”并将结果压入栈顶

0x82	ixor	将栈顶两 int 型数值作“按位异或”并将结果压入栈顶
0x83	lxor	将栈顶两 long 型数值作“按位异或”并将结果压入栈顶
0x84	iinc M N	（M 为非负整数，N 为整数）将局部变量数组的第 M 个单元中的 int 值增加 N，常用于 for 循环中自增量的更新
0x85	i2l	将栈顶 int 型数值强制转换成 long 型数值，并将结果压入栈顶
0x86	i2f	将栈顶 int 型数值强制转换成 float 型数值，并将结果压入栈顶
0x87	i2d	将栈顶 int 型数值强制转换成 double 型数值，并将结果压入栈顶
0x88	l2i	将栈顶 long 型数值强制转换成 int 型数值，并将结果压入栈顶
0x89	l2f	将栈顶 long 型数值强制转换成 float 型数值，并将结果压入栈顶
0x8a	l2d	将栈顶 long 型数值强制转换成 double 型数值，并将结果压入栈顶
0x8b	f2i	将栈顶 float 型数值强制转换成 int 型数值，并将结果压入栈顶
0x8c	f2l	将栈顶 float 型数值强制转换成 long 型数值，并将结果压入栈顶
0x8d	f2d	将栈顶 float 型数值强制转换成 double 型数值，并将结果压入栈顶
0x8e	d2i	将栈顶 double 型数值强制转换成 int 型数值，并将结果压入栈顶
0x8f	d2l	将栈顶 double 型数值强制转换成 long 型数值，并将结果压入栈顶
0x90	d2f	将栈顶 double 型数值强制转换成 float 型数值，并将结果压入栈顶
0x91	i2b	将栈顶 int 型数值强制转换成 byte 型数值，并将结果压入栈顶
0x92	i2c	将栈顶 int 型数值强制转换成 char 型数值，并将结果压入栈顶
0x93	i2s	将栈顶 int 型数值强制转换成 short 型数值，并将结果压入栈顶
0x94	lcmp	比较栈顶两 long 型数值的大小，并将结果（1、0 或 -1）压入栈顶
0x95	fcmpl	比较栈顶两 float 型数值的大小，并将结果（1、0 或 -1）压入栈顶；当其中一个数值为 “NaN” 时，将 **-1** 压入栈顶
0x96	fcmpg	比较栈顶两 float 型数值的大小，并将结果（1、0 或 -1）压入栈顶；当其中一个数值为 “NaN” 时，将 **1** 压入栈顶
0x97	dcmpl	比较栈顶两 double 型数值的大小，并将结果（1、0 或 -1）压入栈顶；当其中一个数值为 “NaN” 时，将 **-1** 压入栈顶
0x98	dcmpg	比较栈顶两 double 型数值的大小，并将结果（1、0 或 -1）压入栈顶；当其中一个数值为 “NaN” 时，将 **1** 压入栈顶
0x99	ifeq	当栈顶 int 型数值等于 0 时跳转
0x9a	ifne	当栈顶 int 型数值不等于 0 时跳转

0x9b	iflt	当栈顶 int 型数值小于 0 时跳转
0x9c	ifge	当栈顶 int 型数值大于或等于 0 时跳转
0x9d	ifgt	当栈顶 int 型数值大于 0 时跳转
0x9e	ifle	当栈顶 int 型数值小于或等于 0 时跳转
0x9f	if_icmpeq	比较栈顶两 int 型数值的大小，当结果等于 0 时跳转
0xa0	if_icmpne	比较栈顶两 int 型数值的大小，当结果不等于 0 时跳转
0xa1	if_icmplt	比较栈顶两 int 型数值的大小，当结果小于 0 时跳转
0xa2	if_icmpge	比较栈顶两 int 型数值的大小，当结果大于或等于 0 时跳转
0xa3	if_icmpgt	比较栈顶两 int 型数值的大小，当结果大于 0 时跳转
0xa4	if_icmple	比较栈顶两 int 型数值的大小，当结果小于或等于 0 时跳转
0xa5	if_acmpeq	比较栈顶两 引用 型数值，当结果相等时跳转
0xa6	if_acmpne	比较栈顶两 引用 型数值，当结果不相等时跳转
0xa7	goto	无条件跳转
0xa8	jsr	跳转至指定的 16 位 offset 位置，并将 jsr 的下一条指令地址压入栈顶
0xa9	ret	返回至局部变量指定的 index 的指令位置（一般与 jsr 或 jsr_w 联合使用）
0xaa	tableswitch	用于 switch 条件跳转，case 值连续（可变长度指令）
0xab	lookupswitch	用于 switch 条件跳转，case 值不连续（可变长度指令）
0xac	ireturn	从当前方法返回 int
0xad	lreturn	从当前方法返回 long
0xae	freturn	从当前方法返回 float
0xaf	dreturn	从当前方法返回 double
0xb0	areturn	从当前方法返回对象引用
0xb1	return	从当前方法返回 void
0xb2	getstatic	获取指定类的静态字段，并将其压入栈顶
0xb3	putstatic	为指定类的静态字段赋值
0xb4	getfield	获取指定类的实例字段，并将其压入栈顶
0xb5	putfield	为指定类的实例字段赋值

0xb6	invokevirtual	调用实例方法
0xb7	invokespecial	调用超类构造方法，实例初始化方法，私有方法
0xb8	invokestatic	调用静态方法
0xb9	invokeinterface	调用接口方法
0xba	--	无此指令
0xbb	new	创建一个对象，并将其引用值压入栈顶
0xbc	newarray	创建一个指定的原始类型（如 int、float、char 等）的数组，并将其引用值压入栈顶
0xbd	anewarray	创建一个引用型（如类、接口、数组 等）的数组，并将其引用值压入栈顶
0xbe	arraylength	获得数组的长度值并将其压入栈顶
0xbf	athrow	将栈顶的异常抛出
0xc0	checkcast	校验类型转换，校验未通过将抛出 ClassCastException
0xc1	instanceof	校验对象是否是指定的类的实例，如果是则将 1 压入栈顶，否则将 0 压入栈顶
0xc2	monitorenter	获得对象的锁，用于同步方法或同步块
0xc3	monitorexit	释放对象的锁，用于同步方法或同步块
0xc4	wide	扩展局部变量的宽度
0xc5	multianewarray	创建指定类型和指定维度的多维数组（执行该指定时，操作数栈中必须包含各维度的长度），并将其引用值压入栈顶
0xc6	ifnull	为 null 时跳转
0xc7	ifnonnull	不为 null 时跳转
0xc8	goto_w	无条件跳转（宽索引）
0xc9	jsr_w	跳转至指定的 32 位 offset 位置，并将 jsr_w 的下一条指令地址压入栈顶

基本数据类型

1、除了long和double类型外，每个变量都占局部变量区中的一个变量槽(slot)，而long及double会占用两个连续的变量槽。

2、大多数对于boolean、byte、short和char类型数据的操作，都使用相应的int类型作为运算类型。

表 6-31　Java 虚拟机指令集所支持的数据类型

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload

（续）

opcode	byte	short	int	long	float	double	char	reference
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

一、加载和存储指令（总）

1、将一个【局部变量表】加载到【操作数栈】：

```
iload、iload_<n>、lload、lload_<n>、fload、fload_<n>、dload、dload_<n>、
aload、aload_<n>
```

2、将一个数值从【操作数栈】存储到【局部变量表】：

```
istore、istore_<n>、lstore、lstore_<n>、fstore、fstore_<n>、dstore、dstore_<n>、astore、astore_<n>
```

3、将一个【常量】加载到【操作数栈】：

```
bipush、sipush、
ldc、ldc_w、ldc2_w、
aconst_null、iconst_m1、iconst_<i>、lconst_<l>、fconst_<f>、dconst_<d>
```

4、扩充局部变量表的访问索引的指令：

```
wide_<n>:_0、_1、_2、_3,
```

存储数据的操作数栈和局部变量表主要就是由加载和存储指令进行操作，除此之外，还有少量指令，如访问对象的字段或数组元素的指令也会向操作数栈传输数据。

二、const系列（小数值）

该系列命令主要负责把【简单的数值类型】送到【操作数栈栈顶】。该系列命令不带参数。注意只把简单的数值类型送到栈顶时，才使用如下的命令。

> 比如对应int型，该方式只能把-1，0，1，2，3，4，5（分别采用iconst_m1，iconst_0，iconst_1，iconst_2，iconst_3，iconst_4，iconst_5）送到栈顶。

对于int型，其他的数值请使用push系列命令（比如bipush）。

指令码	助记符	说明
0x02	iconst_m1	将int型(-1)推送至栈顶
0x03	iconst_0	将int型(0)推送至栈顶
0x04	iconst_1	将int型(1)推送至栈顶
0x05	iconst_2	将int型(2)推送至栈顶
0x06	iconst_3	将int型(3)推送至栈顶
0x07	iconst_4	将int型(4)推送至栈顶
0x08	iconst_5	将int型(5)推送至栈顶
0x09	lconst_0	将long型(0)推送至栈顶
0x0a	lconst_1	将long型(1)推送至栈顶

0x0b	fconst_0	将float型(0)推送至栈顶
0x0c	fconst_1	将float型(1)推送至栈顶
0x0d	fconst_2	将float型(2)推送至栈顶
0x0e	dconst_0	将double型(0)推送至栈顶
0x0f	dconst_1	将double型(1)推送至栈顶

三、push系列（中数值）

该系列命令负责把一个【整形数字（长度比较小）】送到到【操作数栈栈顶】。该系列命令【有一个参数】，用于指定要送到栈顶的数字。

> 注意该系列命令只能操作一定范围内的整形数值，超出该范围的使用将使用【ldc命令】系列。

指令码	助记符	说明
0x10	bipush	将单字节的常量值(-128~127)推送至栈顶
0x11	sipush	将一个短整型常量值(-32768~32767)推送至栈顶

四、ldc系列（大数值或字符串常量）

该系列命令负责把【长度较长的数值常量】或【String常量值】从【常量池中】推送至【操作数栈栈顶】。该命令后面需要给一个表示常量在常量池中位置(编号)的参数，

哪些常量是放在常量池呢？比如：

```
final static int id=32768;
final static float double=6.5
```

> 对于const系列命令和push系列命令操作范围之外的数值类型常量，都放在常量池中。

另外，所有不是通过new创建的String都是放在常量池中的。

指令码	助记符	说明
0x12	ldc	将int， float或String型常量值从常量池中推送至栈顶
0x13	ldc_w	将int， float或String型常量值从常量池中推送至栈顶（宽索引）
0x14	ldc2_w	将long或double型常量值从常量池中推送至栈顶（宽索引）

五、load系列

5.1、load系列A

该系列命令负责把【本地变量表中的值】送到【操作数栈栈顶】。这里的本地变量不仅可以是数值类型，还可以是引用类型。

> - 对于前四个本地变量可以采用iload_0，iload_1，iload_2，iload_3(它们分别表示第0，1，2，3个整形变量)这种不带参数的简化命令形式。
> - 对于第4以上的本地变量将使用iload命令这种形式，在它后面给一参数，以表示是对第几个(从0开始)本类型的本地变量进行操作。对本地变量所进行的编号，是对所有类型的本地变量进行的（并不按照类型分类）。
> - 对于非静态函数（虚方法，实例方法），第一变量是this，即其对应的操作是aload_0。
> - 还有函数传入参数也算本地变量，在进行编号时，它是先于函数体的本地变量的。

指令码	助记符	说明
0x15	iload	将指定的int型本地变量推送至栈顶
0x16	lload	将指定的long型本地变量推送至栈顶
0x17	fload	将指定的float型本地变量推送至栈顶
0x18	dload	将指定的double型本地变量推送至栈顶
0x19	aload	将指定的引用类型本地变量推送至栈顶
0x1a	iload_0	将第一个int型本地变量推送至栈顶
0x1b	iload_1	将第二个int型本地变量推送至栈顶
0x1c	iload_2	将第三个int型本地变量推送至栈顶
0x1d	iload_3	将第四个int型本地变量推送至栈顶
0x1e	lload_0	将第一个long型本地变量推送至栈顶
0x1f	lload_1	将第二个long型本地变量推送至栈顶
0x20	lload_2	将第三个long型本地变量推送至栈顶
0x21	lload_3	将第四个long型本地变量推送至栈顶
0x22	fload_0	将第一个float型本地变量推送至栈顶
0x23	fload_1	将第二个float型本地变量推送至栈顶
0x24	fload_2	将第三个float型本地变量推送至栈顶

0x25	fload_3	将第四个float型本地变量推送至栈顶
0x26	dload_0	将第一个double型本地变量推送至栈顶
0x27	dload_1	将第二个double型本地变量推送至栈顶
0x28	dload_2	将第三个double型本地变量推送至栈顶
0x29	dload_3	将第四个double型本地变量推送至栈顶
0x2a	aload_0	将第一个引用类型本地变量推送至栈顶
0x2b	aload_1	将第二个引用类型本地变量推送至栈顶
0x2c	aload_2	将第三个引用类型本地变量推送至栈顶
0x2d	aload_3	将第四个引用类型本地变量推送至栈顶

5.2、load系列B

该系列命令负责把数组的某项送到栈顶。该命令根据栈里内容来确定对哪个数组的哪项进行操作。

比如，如果有成员变量：

```
final String names[]={"robin", "hb"};
```

那么这句话：

```
String str=names[0];
```

对应的指令为

```
   17: aload_0      //将this引用推送至栈顶，即压入栈。

   18: getfield #5; //Field names:[Ljava/lang/String;
      //将栈顶的指定的对象的第5个实例域（Field）的值（这个值可能是引用，这里就是引用）压入栈顶

   21: iconst_0     //数组的索引值（下标）推至栈顶，即压入栈

   22: aaload       //根据栈里内容来把name数组的第一项的值推至栈顶

   23: astore 5     //把栈顶的值存到str变量里。因为str在我的程序中是其所在非静态函数的第5个变量(从0开始计数)，
```

指令码	助记符	说明
0x2e	iaload	将int型数组指定索引的值推送至栈顶
0x2f	laload	将long型数组指定索引的值推送至栈顶
0x30	faload	将float型数组指定索引的值推送至栈顶
0x31	daload	将double型数组指定索引的值推送至栈顶
0x32	aaload	将引用型数组指定索引的值推送至栈顶
0x33	baload	将boolean或byte型数组指定索引的值推送至栈顶
0x34	caload	将char型数组指定索引的值推送至栈顶
0x35	saload	将short型数组指定索引的值推送至栈顶

六、store系列

6.1、store系列A

该系列命令负责把【操作数栈栈顶的值】存入【本地变量表】。这里的本地变量不仅可以是数值类型，还可以是引用类型。

- 如果是把栈顶的值存入到前四个本地变量的话，采用的是istore_0，istore_1，istore_2，istore_3(它们分别表示第0，1，2，3个本地整形变量)这种不带参数的简化命令形式。
- 如果是把栈顶的值存入到第四个以上本地变量的话，将使用istore命令这种形式，在它后面给一参数，以表示是把栈顶的值存入到第几个(从0开始)本地变量中。对本地变量所进行的编号，是对所有类型的本地变量进行的（并不按照类型分类）。
- 对于非静态函数，第一变量是this，它是只读的.
- 还有函数传入参数也算本地变量，在进行编号时，它是先于函数体的本地变量的。

指令码	助记符	说明
0x36	istore	将栈顶int型数值存入指定本地变量
0x37	lstore	将栈顶long型数值存入指定本地变量
0x38	fstore	将栈顶float型数值存入指定本地变量
0x39	dstore	将栈顶double型数值存入指定本地变量
0x3a	astore	将栈顶引用型数值存入指定本地变量
0x3b	istore_0	将栈顶int型数值存入第一个本地变量

0x3c	istore_1	将栈顶int型数值存入第二个本地变量
0x3d	istore_2	将栈顶int型数值存入第三个本地变量
0x3e	istore_3	将栈顶int型数值存入第四个本地变量
0x3f	lstore_0	将栈顶long型数值存入第一个本地变量
0x40	lstore_1	将栈顶long型数值存入第二个本地变量
0x41	lstore_2	将栈顶long型数值存入第三个本地变量
0x42	lstore_3	将栈顶long型数值存入第四个本地变量
0x43	fstore_0	将栈顶float型数值存入第一个本地变量
0x44	fstore_1	将栈顶float型数值存入第二个本地变量
0x45	fstore_2	将栈顶float型数值存入第三个本地变量
0x46	fstore_3	将栈顶float型数值存入第四个本地变量
0x47	dstore_0	将栈顶double型数值存入第一个本地变量
0x48	dstore_1	将栈顶double型数值存入第二个本地变量
0x49	dstore_2	将栈顶double型数值存入第三个本地变量
0x4a	dstore_3	将栈顶double型数值存入第四个本地变量
0x4b	astore_0	将栈顶引用型数值存入第一个本地变量
0x4c	astore_1	将栈顶引用型数值存入第二个本地变量
0x4d	astore_2	将栈顶引用型数值存入第三个本地变量
0x4e	astore_3	将栈顶引用型数值存入第四个本地变量

6.2、store系列B

该系列命令负责把栈顶项的值存到数组里。该命令根据栈里内容来确定对哪个数组的哪项进行操作。

比如，如下代码:

```
int moneys[]=new int[5];

moneys[1]=100;
```

其对应的指令为:

```
   49: iconst_5

   50: newarray int

   52: astore 11

   54: aload 11

   56: iconst_1

   57: bipush 100

   59: iastore

   60: lload 6        //因为str在我的程序中是其所非静态在函数的第6个变量(从0开始计数).
```

指令码	助记符	说明
0x4f	iastore	将栈顶int型数值存入指定数组的指定索引位置
0x50	lastore	将栈顶long型数值存入指定数组的指定索引位置
0x51	fastore	将栈顶float型数值存入指定数组的指定索引位置
0x52	dastore	将栈顶double型数值存入指定数组的指定索引位置
0x53	aastore	将栈顶引用型数值存入指定数组的指定索引位置
0x54	bastore	将栈顶boolean或byte型数值存入指定数组的指定索引位置
0x55	castore	将栈顶char型数值存入指定数组的指定索引位置
0x56	sastore	将栈顶short型数值存入指定数组的指定索引位置

七、pop系列

该系列命令似乎只是简单对栈顶进行操作，更多详情待补充。

指令码	助记符	说明
0x57	pop	将栈顶数值弹出 (数值不能是long或double类型的)
0x58	pop2	将栈顶的一个（long或double类型的)或两个数值弹出（其它）
0x59	dup	复制栈顶数值(数值不能是long或double类型的)并将复制值压入栈顶

0x5a	dup_x1	复制栈顶数值(数值不能是long或double类型的)并将两个复制值压入栈顶
0x5b	dup_x2	复制栈顶数值(数值不能是long或double类型的)并将三个（或两个）复制值压入栈顶
0x5c	dup2	复制栈顶一个（long或double类型的)或两个（其它）数值并将复制值压入栈顶
0x5d	dup2_x1	复制栈顶数值(long或double类型的)并将两个复制值压入栈顶
0x5e	dup2_x2	复制栈顶数值(long或double类型的)并将三个（或两个）复制值压入栈顶

八、栈顶元素数学操作及移位操作系列

该系列命令用于对栈顶元素行数学操作，和对数值进行移位操作。移位操作的操作数和要移位的数都是从栈里取得。

比如对于代码：

```
int k=100;k=k>>1;
```

其对应的JVM指令为：

```
   60: bipush 100

   62: istore 12//因为k在我的程序中是其所在非静态函数的第12个变量(从0开始计数).

   64: iload 12

   66: iconst_1

   67: ishr

   68: istore 12
```

指令码	助记符	说明

0x5f	swap	将栈最顶端的两个数值互换(数值不能是long或double类型的)
0x60	iadd	将栈顶两int型数值相加并将结果压入栈顶
0x61	ladd	将栈顶两long型数值相加并将结果压入栈顶
0x62	fadd	将栈顶两float型数值相加并将结果压入栈顶
0x63	dadd	将栈顶两double型数值相加并将结果压入栈顶
0x64	isub	将栈顶两int型数值相减并将结果压入栈顶
0x65	lsub	将栈顶两long型数值相减并将结果压入栈顶
0x66	fsub	将栈顶两float型数值相减并将结果压入栈顶
0x67	dsub	将栈顶两double型数值相减并将结果压入栈顶
0x68	imul	将栈顶两int型数值相乘并将结果压入栈顶
0x69	lmul	将栈顶两long型数值相乘并将结果压入栈顶
0x6a	fmul	将栈顶两float型数值相乘并将结果压入栈顶
0x6b	dmul	将栈顶两double型数值相乘并将结果压入栈顶
0x6c	idiv	将栈顶两int型数值相除并将结果压入栈顶
0x6d	ldiv	将栈顶两long型数值相除并将结果压入栈顶
0x6e	fdiv	将栈顶两float型数值相除并将结果压入栈顶
0x6f	ddiv	将栈顶两double型数值相除并将结果压入栈顶
0x70	irem	将栈顶两int型数值作取模运算并将结果压入栈顶
0x71	lrem	将栈顶两long型数值作取模运算并将结果压入栈顶
0x72	frem	将栈顶两float型数值作取模运算并将结果压入栈顶
0x73	drem	将栈顶两double型数值作取模运算并将结果压入栈顶
0x74	ineg	将栈顶int型数值取负并将结果压入栈顶
0x75	lneg	将栈顶long型数值取负并将结果压入栈顶
0x76	fneg	将栈顶float型数值取负并将结果压入栈顶

0x77	dneg	将栈顶double型数值取负并将结果压入栈顶
0x78	ishl	将int型数值左移位指定位数并将结果压入栈顶
0x79	lshl	将long型数值左移位指定位数并将结果压入栈顶
0x7a	ishr	将int型数值右（符号）移位指定位数并将结果压入栈顶
0x7b	lshr	将long型数值右（符号）移位指定位数并将结果压入栈顶
0x7c	iushr	将int型数值右（无符号）移位指定位数并将结果压入栈顶
0x7d	lushr	将long型数值右（无符号）移位指定位数并将结果压入栈顶
0x7e	iand	将栈顶两int型数值作“按位与”并将结果压入栈顶
0x7f	land	将栈顶两long型数值作“按位与”并将结果压入栈顶
0x80	ior	将栈顶两int型数值作“按位或”并将结果压入栈顶
0x81	lor	将栈顶两long型数值作“按位或”并将结果压入栈顶
0x82	ixor	将栈顶两int型数值作“按位异或”并将结果压入栈顶
0x83	lxor	将栈顶两long型数值作“按位异或”并将结果压入栈顶

运算指令

1、运算或算术指令用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。

2、算术指令分为两种：整型运算的指令和浮点型运算的指令。

3、无论是哪种算术指令，都使用Java虚拟机的数据类型，由于没有直接支持byte、short、char和boolean类型的算术指令，使用操作int类型的指令代替。

```
加法指令：iadd、ladd、fadd、dadd。
减法指令：isub、lsub、fsub、dsub。
乘法指令：imul、lmul、fmul、dmul。
除法指令：idiv、ldiv、fdiv、ddiv。
求余指令：irem、lrem、frem、drem。
取反指令：ineg、lneg、fneg、dneg。
位移指令：ishl、ishr、iushr、lshl、lshr、lushr。
按位或指令：ior、lor。
按位与指令：iand、land。
按位异或指令：ixor、lxor。
局部变量自增指令：iinc。
比较指令：dcmpg、dcmpl、fcmpg、fcmpl、lcmp。
```

类型转换指令

1、类型转换指令可以将两种不同的数值类型进行相互转换。

2、这些转换操作一般用于实现用户代码中的显式类型转换操作，或者用来处理字节码指令集中数据类型相关指令无法与数据类型一一对应的问题。

宽化类型转换

int类型到long、float或者double类型。
long类型到float、double类型。
float类型到double类型。

```
i2l、f2b、l2f、l2d、f2d。
```

窄化类型转换

```
i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l和d2f。
```

对象创建与访问指令

创建类实例的指令：new。
创建数组的指令：newarray、anewarray、multianewarray。
访问类字段（static字段，或者称为类变量）和实例字段（非static字段，或者称为实例变量）的指令：getfield、putfield、getstatic、putstatic。
把一个数组元素加载到操作数栈的指令：baload、caload、saload、iaload、laload、faload、daload、aaload。
将一个操作数栈的值存储到数组元素中的指令：bastore、castore、sastore、iastore、fastore、dastore、aastore。
取数组长度的指令：arraylength。
检查类实例类型的指令：instanceof、checkcast。

操作数栈管理指令

直接操作操作数栈的指令：

将操作数栈的栈顶一个或两个元素出栈：pop、pop2。
复制栈顶一个或两个数值并将复制值或双份的复制值重新压入栈顶：dup、dup2、dup_x1、dup2_x1、dup_x2、dup2_x2。
将栈最顶端的两个数值互换：swap。

控制转移指令

1、控制转移指令可以让Java虚拟机有条件或无条件地从指定的位置指令而不是控制转移指令的下一条指令继续执行程序。

2、从概念模型上理解，可以认为控制转移指令就是在有条件或无条件地修改PC寄存器的值。

条件分支：ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull、if_icmpeq、if_icmpne、if_icmplt、if_icmpgt、if_icmple、if_icmpge、if_acmpeq和if_acmpne。
复合条件分支：tableswitch、lookupswitch。
无条件分支：goto、goto_w、jsr、jsr_w、ret。

在Java虚拟机中有专门的指令集用来处理int和reference类型的条件分支比较操作，为了可以无须明显标识一个实体值是否null，也有专门的指令用来检测null值。

方法执行

以下面代码为例看一下执行引擎是如何将一段代码在执行部件上执行的，如下一段【Java源代码】：

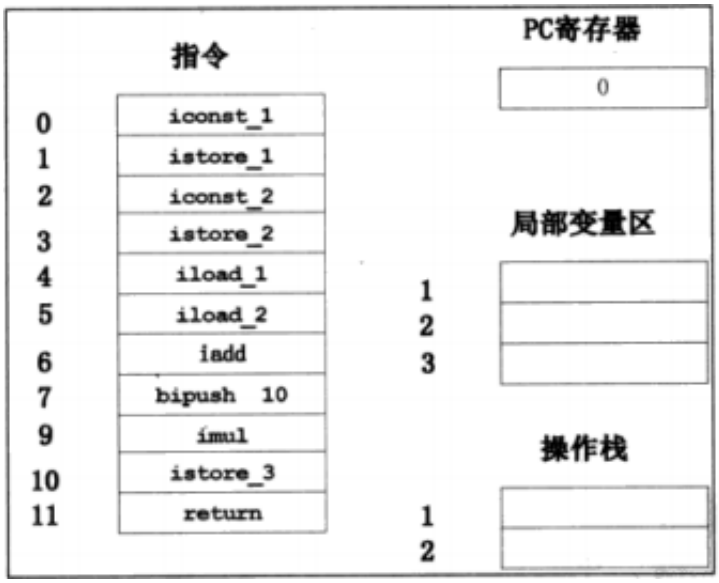
```
public  class Math{
     public static void main(String[] args){
           int a = 1 ;
           int b = 2;
           int c = (a+b)*10;
     }
}
```

其中main的【class字节码】指令如下：

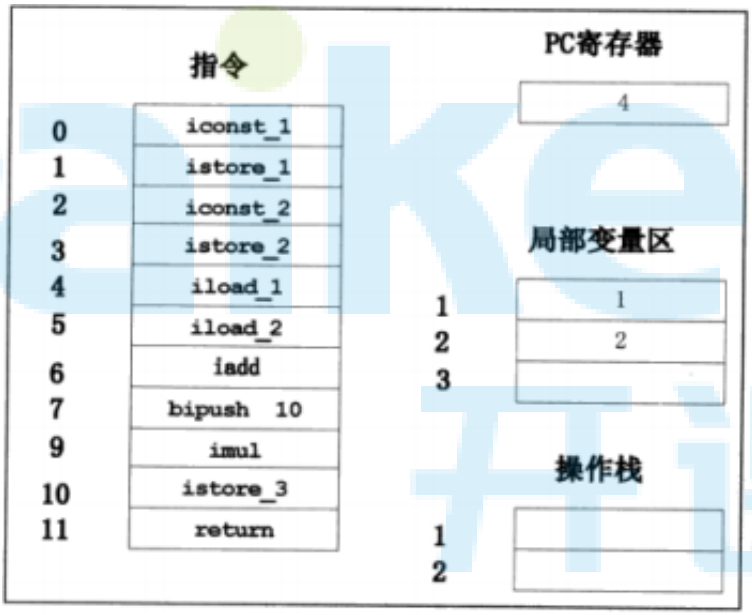
偏移量	指令	说明
0：	iconst_1	常数1入栈
1：	istore_1	将栈顶元素移入本地变量1存储
2：	iconst_2	常数2入栈
3：	istore_2	将栈顶元素移入本地变量2存储
4：	iload_1	本地变量1入栈
5：	iload_2	本地变量2入栈

```
6:      iadd          弹出栈顶两个元素相加
7:      bipush 10      将10入栈
9:      imul          栈顶两个元素相乘
10:     istore_3       栈顶元素移入本地变量3存储
11:     return         返回
```

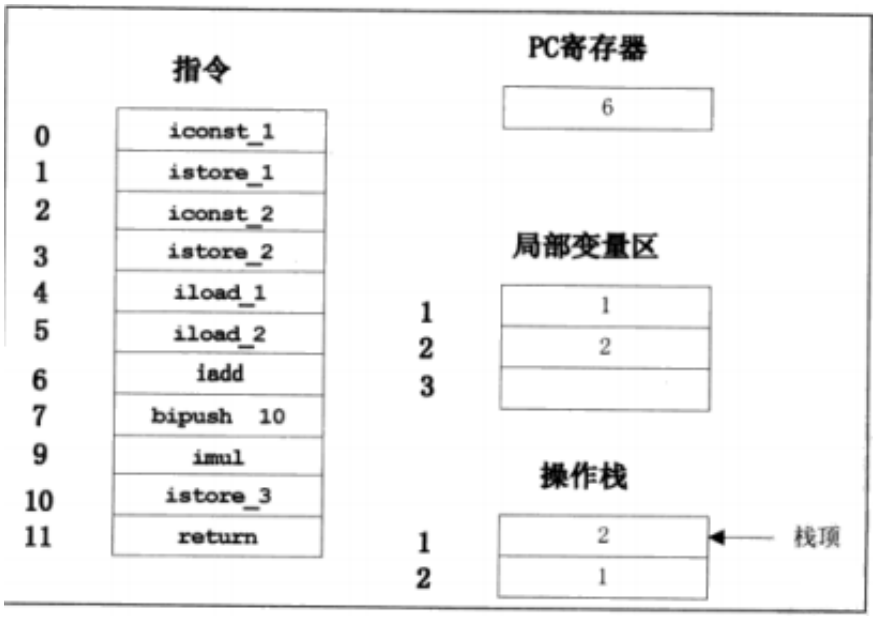

对应到执行引擎的各执行部件如图:



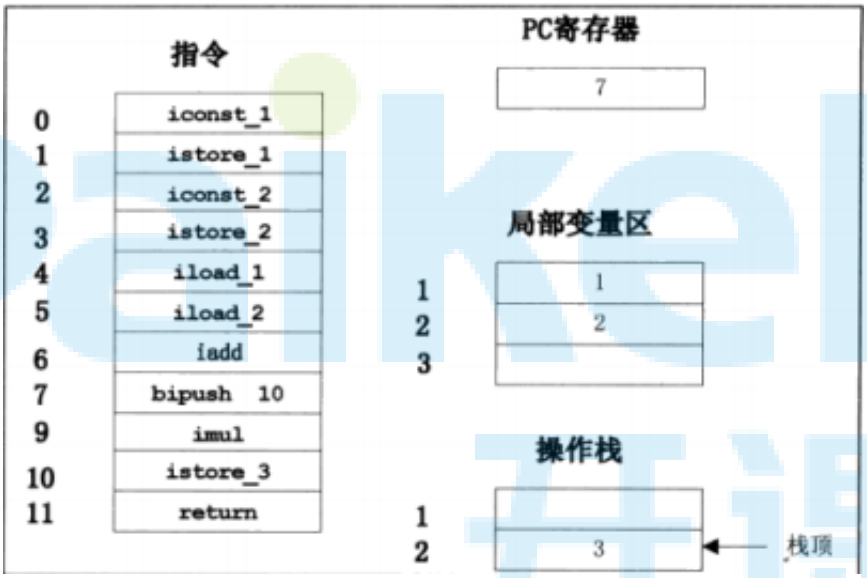
在开始执行方法之前，PC寄存器存储的指针是第1条指令的地址，局部变量区和操作栈都没有数据。从第1条到第4条指令分别将a、b两个本地变量赋值，对应到局部变量区就是1和2分别存储常数1和2，如图：



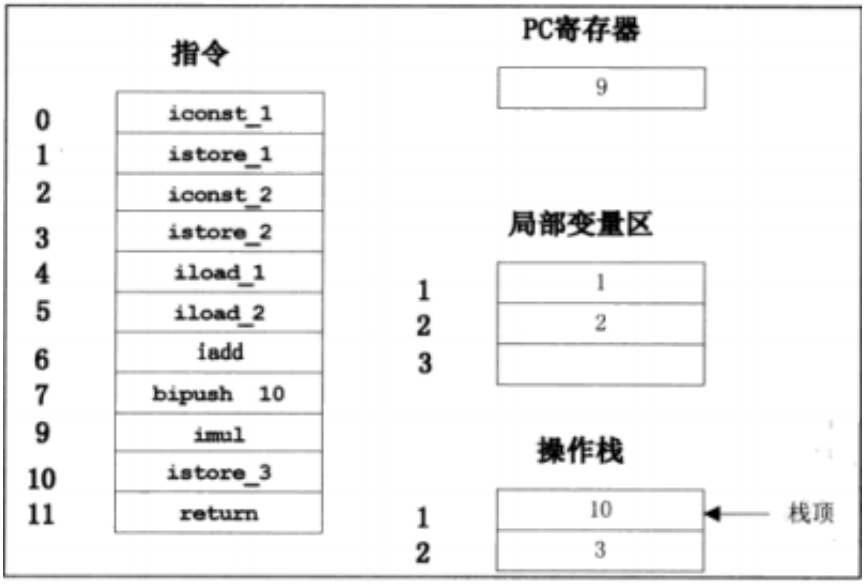
第5条和第6条指令分别是将两个局部变量入栈，然后相加，如图：



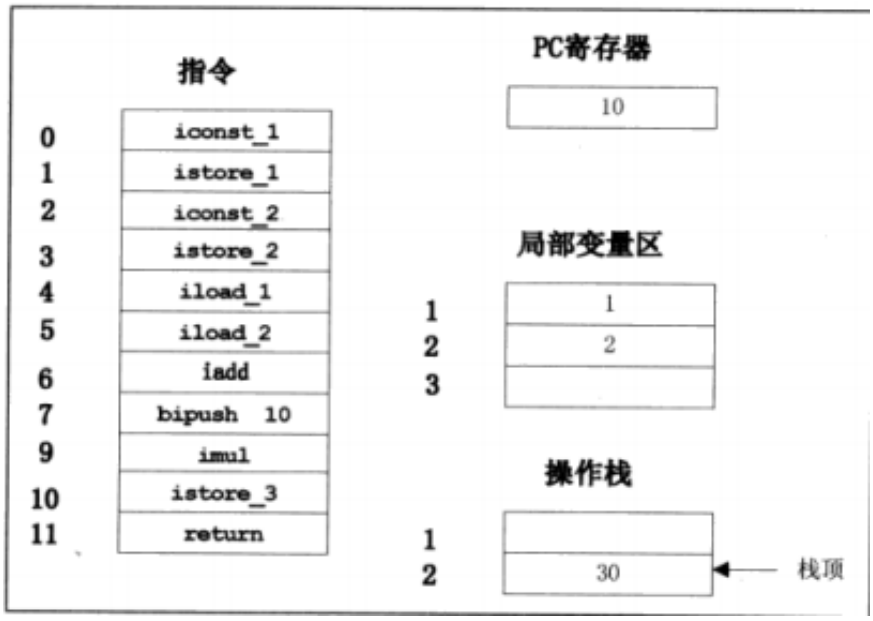
1先入栈2后入栈，栈顶元素是2，第7条指令是将栈顶的两个元素弹出后相加，结果再入栈，如图：



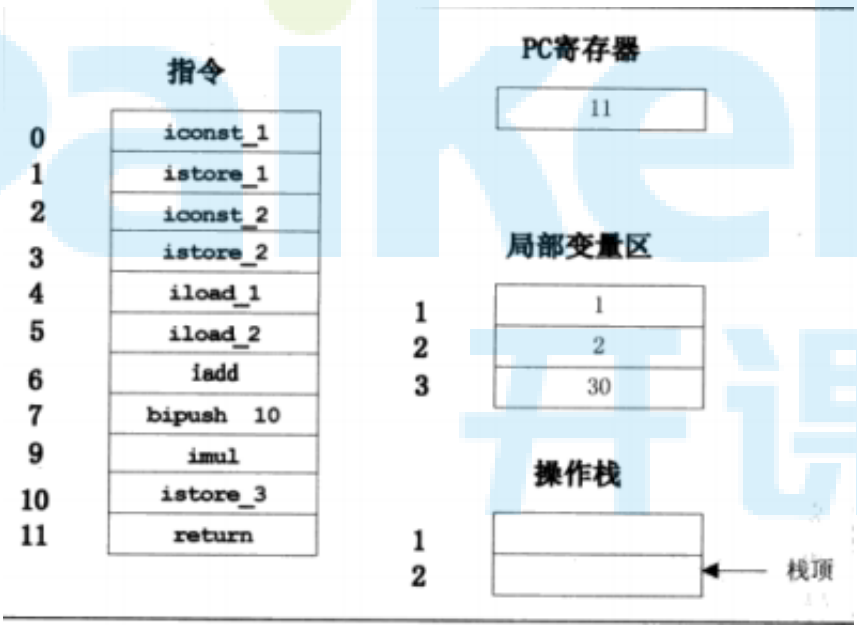
可以看出，变量a和b相加的结果3存在当前栈顶中，接下来第8条指令将10入栈，如图：



当前PC寄存器执行的地址是9，下一个操作是将当前栈的两个操作数弹出进行相乘并把结果压入栈中，如图：



第10条指令是将当前的栈顶元素存入局部变量3中，如图：



第10条指令执行完后栈中元素出栈，出栈的元素存储在局部变量区3中，对应的是变量c的值。最后一条指令是return，这条指令执行完后当前的这个方法对应的这些部件会被JVM回收，局部变量区的所有值将全部释放，PC寄存器会被销毁，在Java栈中与这个方法对应的栈帧将消失。

五、方法调用

静态绑定和动态绑定

在JVM中，将符号引用替换为直接引用（目标方法的内存地址）的处理逻辑，与方法的绑定机制有关系。

Java语言具有继承和多态的特性，所以自然而然的也就具备以下两种绑定方式：静态绑定和动态绑定。

当程序中调用了某一个方法时：

- 如果在编译期（直接可以根据编译类型确定内存中的目标方法）就可以确定目标方法的话，这其实就是一种静态绑定机制，也叫静态链接，也叫静态分派，也叫前期绑定。
    - 静态方法
    - 私有方法
    - final方法
    - this()
    - super()
- 如果在运行期（不能根据编译类型确定内存中的目标方法，需要根据对象类型去确定内存中的目标方法）间才能根据对象类型确定目标方法的话，这是动态绑定机制，也叫动态链接、也叫动态分派，也叫后期绑定。

虚方法和非虚方法

如果在编译期就可以确定的目标方法，就被称为非虚方法（静态绑定）。反之都是虚方法（动态绑定）。

非虚方法包括：静态方法、私有方法、final方法、实例构造器、父类方法

方法调用指令

我们再从JVM层面分析下，JVM里面是通过哪里指令来实现方法的调用的：

- invokestatic:调用静态方法
- invokespecial:调用非静态私有方法、构造方法(包括super)

- invokeinterface:调用接口方法([多态]())
- invokevirtual:调用非静态非私有方法([多态]())

- invokedynamic:动态调用（Java7引入的，第一次用却是在Java8中，用在了Lambda表达式和默认方法中，它允许调用任意类中的同名方法，注意是任意类，和重载重写不同）(动态 ≠ 多态)

语言分为静态语言和动态语言。

- 静态语言：编译期间就确定变量类型。Java是静态语言（JDK8之后，使用lambda表达式去实现动态语言功能）
- 动态语言：运行请求才能确定变量类型。JS

具体来说，Java字节码指令中与调用相关的指令共有五种：

- invokevirtual：

  用于调用对象的实例方法，根据对象的实际类型进行分派（虚方法分派），这也是Java语言中最常见的方法分派方式。

- invokeinterface：

  用于调用接口方法，它会在运行时搜索一个实现了这个接口方法的对象，找出适合的方法进行调用。

- invokespecial：

  用于调用一些需要特殊处理的实例方法，包括实例初始化（<init>）方法、私有方法和父类方法。

- invokestatic：

  调用静态方法（static方法）。

- invokedynamic：

  用于在运行时动态解析出调用点限定符所引用的方法，并执行该方法，前面4条调用指令的分派逻辑都固化在Java虚拟机内部，而invokedynamic指令的分派逻辑是由用户所设定的引导方法决定的。

```
interface Student {
  boolean isRecommend();
}

class Edu {
  public double youhui (double originPrice,  Student stu) {
    return originPrice * 0.7d;
  }
}

class Kaikeba extends Edu {
  @Override
  public double youhui (double originPrice,  Student stu) {
    if (stu.isRecommend()) {                    // invokeinterface
      return originPrice * randomYouhui ();     // invokestatic
    } else {
      return super.youhui(originPrice,  stu);   // invokespecial
    }
  }
  private static double randomYouhui () {
```

```
    return new Random()                                    // invokespecial
            .nextDouble()    ;                             // invokevirtual
  }
}
```

重载方法的查找过程演示

重载方法的查找过程是发生在编译过程中的。重载方法又叫编译时多态。

- 根据编译类型去查找方法名称和方法声明（参数类型和返回值）
- 如果没有找到，则查找是否有凑合的方法（可以自动类型转换的参数）
- 实在找不到，则报错

```
static abstract class Human{}
static class Man extends Human{ }
static class Woman extends Human{}

public void sayHello(Human guy){
    System.out.println("hello,人类! ");//1
}
public void sayHello(Man guy){
    System.out.println("hello,老铁! ");//2
}
public void sayHello(Woman guy){
    System.out.println("hello,老妹! ");//3
}

public static void main(String[]args){

    Human h1 = new Man();
    Human h2 = new Woman();

    StaticCall02 sd = new StaticCall02();
    sd.sayHello(h1);
    sd.sayHello(h2);


}
```

发生了方法的重载，方法重载会在编译期确定具体调用哪个方法，这个也相当于是一种静态分派

编译看左边
运行看右边

编译时，确定的h1类型是左边的类型，也就是Human类型

运行时多态查找虚方法的过程

```
package com.kkb.test;

//调用方法
public class AutoCall {
  public static void main(String[] args) {
```

```
    Father father = new Son();
    // 多态
    father.f1();
    // 打印结果：  Son-f1()
  }
}

// 被调用的父类
class Father {
  public void f1() {
    System.out.println("father-f1()");
  }

  public void f1(int i) {
    System.out.println("father-f1() para-int " + i);
  }
}

// 被调用的子类
class Son extends Father {
  public void f1() {
    // 覆盖父类的方法
    System.out.println("Son-f1()");
  }

  public void f1(char c) {
    System.out.println("Son-s1() para-char " + c);
  }
}
```

```java
// 被调用的父类
class Father {
  public void f1() {
    System.out.println("father-f1()");
  }

  public void f1(int i) {
    System.out.println("father-f1() para-int " + i);
  }
}

// 被调用的子类
class Son extends Father {
  public void f1() {
    // 覆盖父类的方法
    System.out.println("Son-f1()");
  }

  public void f1(char c) {
    System.out.println("Son-s1() para-char " + c);
  }
}
```

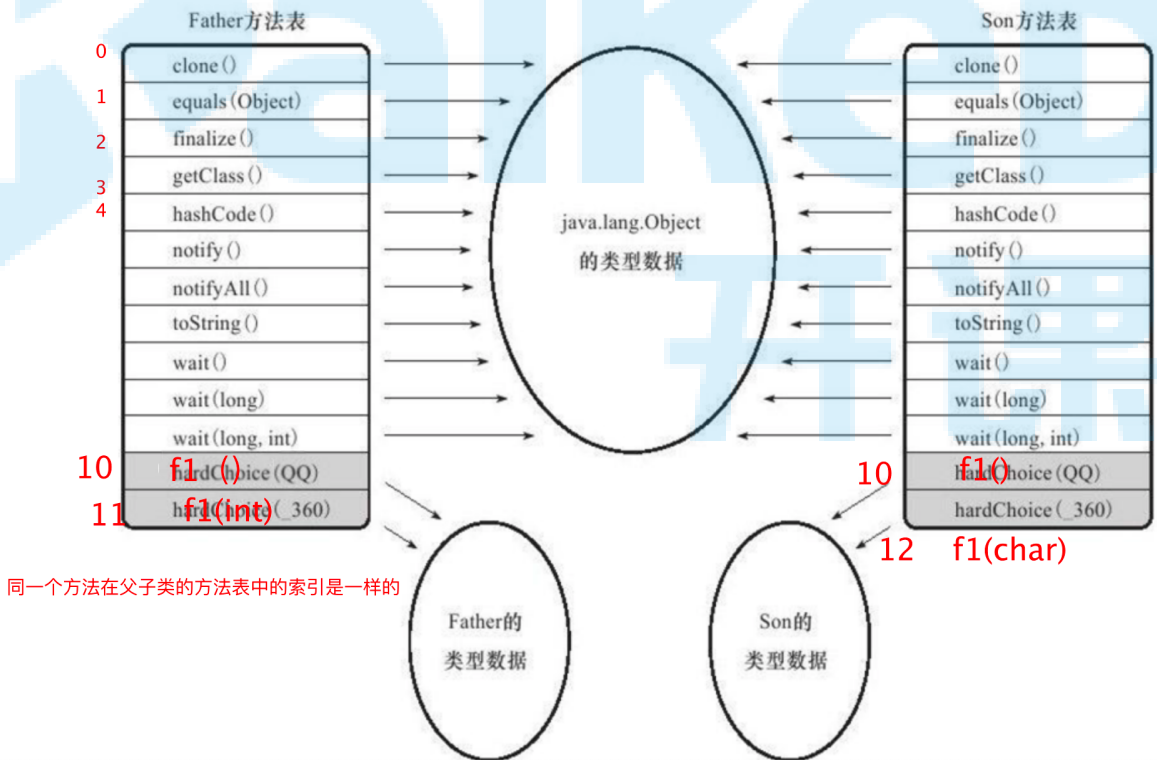
```
PS D:\05-workspace\vip-class\jvm\bin\com\kkb\test> javap -v .\AutoCall
警告: 二进制文件.\AutoCall包含com.kkb.test.AutoCall
Classfile /D:/05-workspace/vip-class/jvm/bin/com/kkb/test/AutoCall.class
  Last modified 2020-4-11; size 553 bytes
  MD5 checksum bb066cb8ac1c23ea2281bae481f39419
  Compiled from "AutoCall.java"
public class com.kkb.test.AutoCall
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Class              #2             // com/kkb/test/AutoCall
   #2 = Utf8               com/kkb/test/AutoCall
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Utf8               Code
   #8 = Methodref          #3.#9          // java/lang/Object."<init>":()V
   #9 = NameAndType        #5:#6          // "<init>":()V
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               Lcom/kkb/test/AutoCall;
```

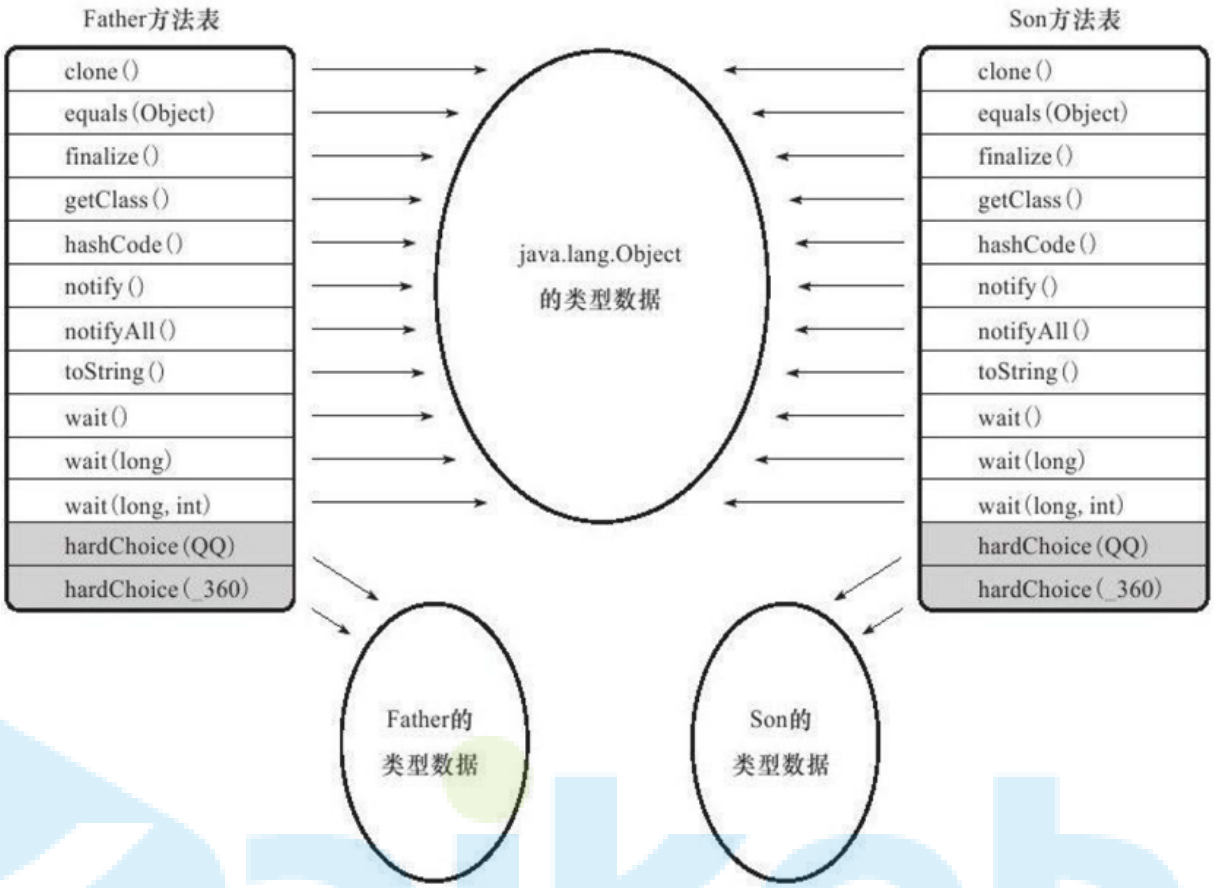
```
  #14 = Utf8               main
  #15 = Utf8               ([Ljava/lang/String;)V
  #16 = Class              #17            // com/kkb/test/Son
  #17 = Utf8               com/kkb/test/Son
  #18 = Methodref          #16.#9         // com/kkb/test/Son."<init>":()V
  #19 = Methodref          #20.#22        // com/kkb/test/Father.f1:()V
  #20 = Class              #21            // com/kkb/test/Father
  #21 = Utf8               com/kkb/test/Father
  #22 = NameAndType        #23:#6         // f1:()V
  #23 = Utf8               f1
  #24 = Utf8               args
  #25 = Utf8               [Ljava/lang/String;
  #26 = Utf8               father
  #27 = Utf8               Lcom/kkb/test/Father;
  #28 = Utf8               MethodParameters
  #29 = Utf8               SourceFile
  #30 = Utf8               AutoCall.java
{
  public com.kkb.test.AutoCall();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #8                  // Method java/lang/Object."
<init>":()V
         4: return
      LineNumberTable:
        line 4: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/kkb/test/AutoCall;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
         0: new           #16                 // class com/kkb/test/Son
         3: dup
         4: invokespecial #18                 // Method com/kkb/test/Son."
<init>":()V
         7: astore_1
         8: aload_1
         9: invokevirtual #19                 // Method com/kkb/test/Father.f1:
()V
        12: return
      LineNumberTable:
        line 6: 0
```

```
        line 8: 8
        line 10: 12
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      13     0  args   [Ljava/lang/String;
            8       5     1 father   Lcom/kkb/test/Father;
    MethodParameters:
      Name                           Flags
      args
}
SourceFile: "AutoCall.java"
```

上面的源代码中有三个重要的概念：多态(polymorphism)、方法覆盖、方法重载。打印的结果大家也都比较清楚，但是JVM是如何知道father.f1()调用的是子类Son中方法而不是Father中的方法呢?在解释这个问题之前，我们首先简单的讲下JVM管理的一个非常重要的数据结构——虚方法表（virtual method table）。

在JVM加载类的同时，会在方法区中为这个类存放很多信息。其中就有一个数据结构叫虚方法表。它以数组的形式记录了当前类及其所有超类的可见方法字节码在内存中的直接地址。





虚方法表有两个特点：

> (1) 子类方法表中继承了父类的方法，比如Father extends Object。
>
> (2) 相同的方法(相同的方法签名：方法名和参数列表)在所有类的方法表中的索引相同。比如Father方法表中的f1()和Son方法表中的f1()都位于各自方法表的第11项中。

对于上面的源代码，编译器首先会把main方法编译成下面的字节码指令：

```
0: new           #16                 // class com/kkb/test/Son
3: dup
4: invokespecial #18                 // Method com/kkb/test/Son."<init>":()V
7: astore_1
8: aload_1
9: invokevirtual #19                 // Method com/kkb/test/Father.f1:()V
12: return
```

其中invokevirtual指令的详细调用过程是这样的：

> (1) invokevirtual指令中的#19指的是AutoCall类的常量池中第19个常量表的索引项。这个常量表(CONSTATN_Methodref_info) 记录的是方法f1信息的符号引用(包括f1所在的类名，方法名和返回类型)。JVM会首先根据这个符号引用找到调用方法f1的类的全限定名: hr.test.Father。这是因为调用方法f1的类的对象father声明为Father类型。

> (2) 在Father类型的虚方法表中查找方法f1，如果找到，则将方法f1在方法表中的索引项11(如上图)记录到AutoCall类的常量池中第15个常量表中(常量池解析)。这里有一点要注意：如果Father类型方法表中没有方法f1，那么即使Son类型中方法表有，编译的时候也通过不了。因为调用方法f1的类的对象father的声明为Father类型。
>
> (3) 在调用invokevirtual指令前有一个aload_1指令，它会将开始创建在堆中的Son对象的引用压入操作数栈。然后invokevirtual指令会根据这个Son对象的引用首先找到堆中的Son对象，然后进一步找到Son对象所属类型的方法表。
>
> (4) 这时通过第(2)步中解析完成的#15常量表中的方法表的索引项11，可以定位到Son类型方法表中的方法f1()，然后通过直接地址找到该方法字节码所在的内存空间。

很明显，根据对象(father)的声明类型(Father)还不能够确定调用方法f1的位置，必须根据father在堆中实际创建的对象类型Son来确定f1方法所在的位置。这种在程序运行过程中，通过动态创建的对象的方法表来定位方法的方式，我们叫做 动态绑定机制 。

扩展题

上面的过程很清楚的反映出在方法覆盖的多态调用的情况下，JVM是如何定位到准确的方法的。但是下面的调用方法JVM是如何定位的呢?(仍然使用上面代码中的Father和Son类型)

```
public class AutoCall{
    public static void main(String[] args){
        Father father=new Son();
        char c='a';
        father.f1(c);
        //打印结果：father-f1() para-int 97
    }
}

// 被调用的父类
class Father {
  public void f1() {
    System.out.println("father-f1()");
  }

  public void f1(int i) {
    System.out.println("father-f1() para-int " + i);
  }
}

// 被调用的子类
class Son extends Father {
  public void f1() {
    // 覆盖父类的方法
    System.out.println("Son-f1()");
  }
```

```
    public void f1(char c) {
        System.out.println("Son-s1() para-char " + c);
    }
}
```

问题是Fahter类型中并没有方法签名为f1(char)的方法呀。但打印结果显示JVM调用了Father类型中的f1(int)方法，并没有调用到Son类型中的f1(char)方法。

根据上面详细阐述的调用过程，首先可以明确的是：JVM首先是根据对象father声明的类型Father来解析常量池的(也就是用Father方法表中的索引项来代替常量池中的符号引用)。如果Father中没有匹配到"合适"的方法，就无法进行常量池解析，这在编译阶段就通过不了。

那么什么叫"合适"的方法呢?当然，方法签名完全一样的方法自然是合适的。但是如果方法中的参数类型在声明的类型中并不能找到呢?比如上面的代码中调用father.f1(char)，Father类型并没有f1(char)的方法签名。实际上，JVM会找到一种"凑合"的办法，就是通过 参数的自动转型 来找 到"合适"的 方法。比如char可以通过自动转型成int，那么Father类中就可以匹配到这个方法了。但是还有一个问题，如果通过自动转型发现可以"凑合"出两个方法的话怎么办?比如下面的代码：

```
class Father{
    public void f1(Object o){
        System.out.println("Object");
    }
    public void f1(double[] d){
        System.out.println("double[]");
    }
}

public class Demo{
    public static void main(String[] args) {
        new Father().f1(null); //打印结果：  double[]
    }
}
```

null可以引用于任何的引用类型，那么JVM如何确定"合适"的方法呢。一个很重要的标准就是：

> 如果一个方法可以接受传递给另一个方法的任何参数，那么第一个方法就相对不合适。比如上面的代码: 任何传递给f1(double[])方法的参数都可以传递给f1(Object)方法，而反之却不行，那么f1(double[])方法就更合适。因此JVM就会调用这个更合适的方法。

总结

(1) 所有私有方法、静态方法、构造器及初始化方法都是采用静态绑定机制。在编译器阶段就已经指明了调用方法在常量池中的符号引用，JVM运行的时候只需要进行一次常量池解析即可。

(2) 类对象方法的调用必须在运行过程中采用动态绑定机制。

首先，根据对象的声明类型(对象引用的类型)找到"合适"的方法。具体步骤如下：

① 如果能在声明类型中匹配到方法签名完全一样(参数类型一致)的方法，那么这个方法是最合适的。

② 在第①条不能满足的情况下，寻找可以“凑合”的方法。标准就是通过将参数类型进行自动转型之后再进行匹配。如果匹配到多个自动转型后的方法签名f(A)和f(B)，则用下面的标准来确定合适的方法：传递给f(A)方法的参数都可以传递给f(B)，则f(A)最合适。反之f(B)最合适 。

③ 如果仍然在声明类型中找不到“合适”的方法，则编译阶段就无法通过。

然后，根据在堆中创建对象的实际类型找到对应的方法表，从中确定具体的方法在内存中的位置。

六、垃圾回收

> 垃圾回收一般发生在堆和方法区，也就是线程共享的区域。
>
> 堆和方法区的内存分配，也是通过垃圾收集器去实现的。
>
> 垃圾回收，也是通过垃圾收集器实现的。
>
> 不同的垃圾收集器可能采用不同的垃圾收集算法，去判断对象是否是垃圾对象。

3.1 判断算法

主要是2种：引用计数法和根搜索算法

3.1.1 引用计数法（Reference Counting）

1. 概念

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器都为0的对象就是不可能再被使用的。

2. 优点

实现简单，判断效率高，大部分情况下都是很不错的算法

3. 缺点

Java语言中没有选用引用计数算法来管理内存，其中最主要的原因是它很难解决**对象之间的相互循环引用**的问题。

```
public class Test {

    public static void main(String[] args) {

        MyObject object1 = new MyObject();
        MyObject object2 = new MyObject();

        object1.object = object2;
        object2.object = object1;
```

```
        object1 = null;
        object2 = null;
    }
}

class MyObject{
     MyObject object;
}
```

3.1.2 根搜索算法（GCRoots Tracing）

1. 概念

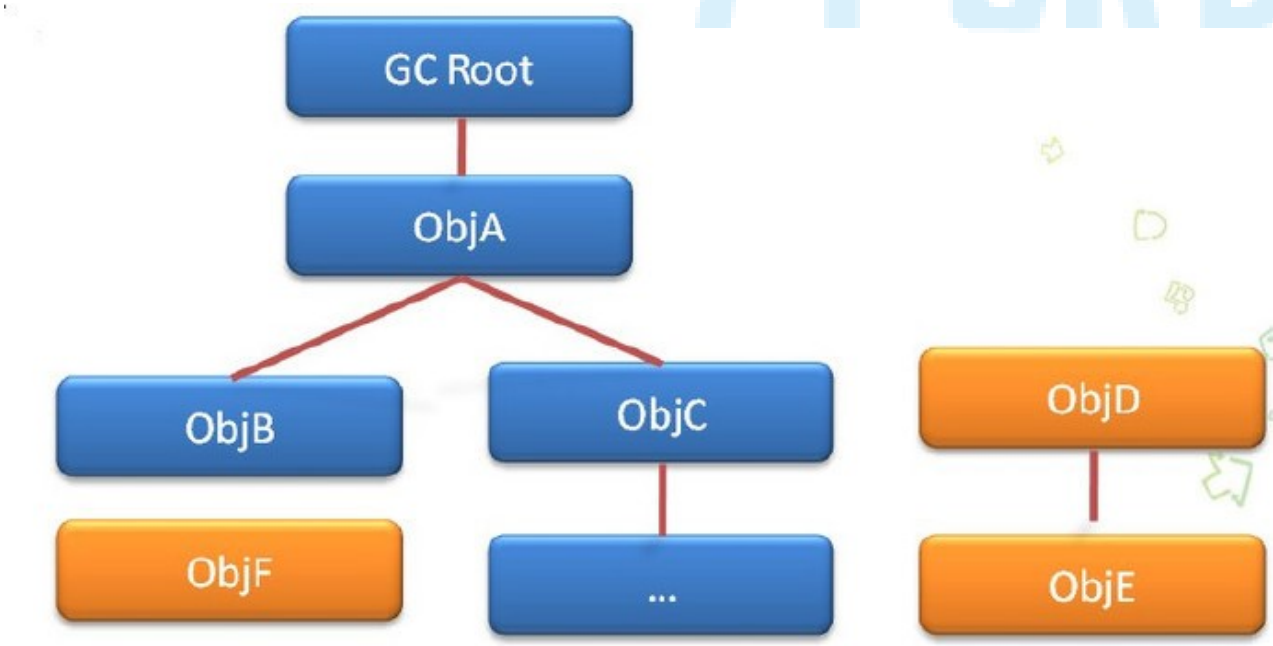
又叫可达性算法。在主流的商用程序语言中（Java和C#），都是使用根搜索算法判定对象是否存活的。

> 基本思路就是通过一系列的名为“GCRoots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GCRoot没有任何引用链相连（就是从GCRoot到这个对象不可达）时，则证明此对象是不可用的。

不可达不一定会被回收，可以用finalize()方法抢救下，但绝对不推荐

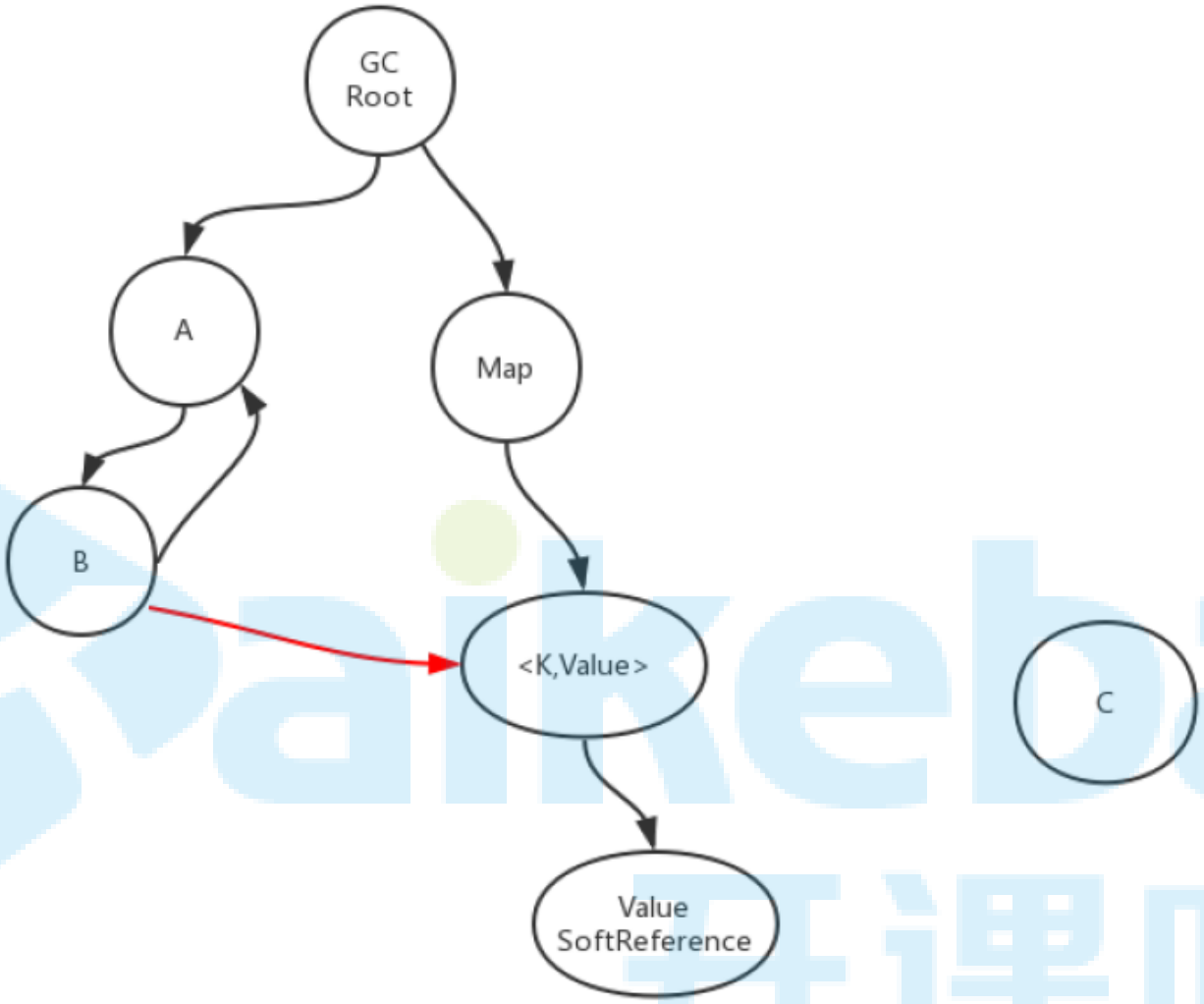
2. 可作GCRoots的对象（堆中引用的不能作为GCRoot对象）

- 虚拟机栈（栈帧中的本地变量表）中的引用的对象。
- 方法区中的类静态属性引用的对象。
- 方法区中的常量引用的对象。
- 本地方法栈中JNI（即一般说的Native方法）的引用的对象。



3.1.3 对象引用（了解）

在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为强引用（StrongReference）、软引用（SoftReference）、弱引用（WeakReference）、虚引用（PhantomReference）四种，这四种引用强度依次逐渐减弱。



1. 强引用

代码中普遍存在，类似“Object obj=new Object()”这类引用，只要强引用还在，就不会被GC。

2. 软引用

非必须引用，内存溢出之前进行回收，如内存还不够，才会抛异常。

```
Object obj = new Object();
SoftReference<Object> sf = new SoftReference<Object>(obj);
obj = null;
sf.get();//有时候会返回null
```

3. 弱引用

非必须引用，只要有GC，就会被回收。

```
Object obj = new Object();
WeakReference<Object> wf = new WeakReference<Object>(obj);
obj = null;
wf.get();//有时候会返回null
wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾
```

弱引用是在第二次垃圾回收时回收，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回null。
弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的isEnQueued方法返回对象是否被垃圾回收器标记。

4. 虚引用

垃圾回收时回收，无法通过引用取到对象值

也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被收集器回收时收到一个系统通知。

```
Object obj = new Object();
PhantomReference<Object> pf = new PhantomReference<Object>(obj);
obj=null;
pf.get();//永远返回null
pf.isEnQueued();//返回是否从内存中已经删除
```

虚引用是每次垃圾回收的时候都会被回收，通过虚引用的get方法永远获取到的数据为null，因此也被成为幽灵引用。
虚引用主要用于检测对象是否已经从内存中删除。

3.1.4 回收过程

即使在可达性分析算法中不可达的对象，也并非是“非死不可”，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程。

- **第一次标记**：如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记；
- **第二次标记**：第一次标记后接着会进行一次筛选，筛选的条件是此对象是否有必要执行`finalize()`方法。在`finalize()`方法中没有重新与引用链建立关联关系的，将被进行第二次标记。

第二次标记成功的对象将真的会被回收，如果对象在`finalize()`方法中重新与引用链建立了关联关系，那么将会逃离本次回收，继续存活。

```
/**
 * 此代码演示了两点：
```

```
 * 1.对象可以在被GC时自我拯救。
 * 2.这种自救的机会只有一次，因为一个对象的finalize()方法最多只会被系统自动调用一次
 */
public class finalizeEscapeGC {

  public static finalizeEscapeGC SAVE_HOOK = null;

  public void isApve() {
      System.out.println("yes, i am still apve :)");
  }

  @Override
  protected void finalize() throws Throwable {
      super.finalize();
      System.out.println("finalize mehtod executed!");
      finalizeEscapeGC.SAVE_HOOK = this;
  }

  public static void main(String[] args) throws Throwable {
      SAVE_HOOK = new finalizeEscapeGC();

      //对象第一次成功拯救自己
      SAVE_HOOK = null;
      System.gc();
      //因为finalize方法优先级很低，所以暂停0.5秒以等待它
      Thread.sleep(500);
      if (SAVE_HOOK != null) {
      SAVE_HOOK.isApve();
      } else {
      System.out.println("no, i am dead :(");
      }

      //下面这段代码与上面的完全相同，但是这次自救却失败了
      SAVE_HOOK = null;
      System.gc();
      //因为finalize方法优先级很低，所以暂停0.5秒以等待它
      Thread.sleep(500);
      if (SAVE_HOOK != null) {
      SAVE_HOOK.isApve();
      } else {
      System.out.println("no, i am dead :(");
      }
  }
}
```

3.1.5 方法区回收（了解）

1. 概念

方法区也是有垃圾回收的，主要回收**废弃常量和无用的类**。

即使满足回收条件也不一定真得回收。主要性价比太低

2. 废弃常量

比如字符串常量，没有对象引用即可回收

常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

3. 无用的类（需要同时满足3个条件）

- 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。
- 加载该类的ClassLoader已经被回收。
- 该类对应的java.lang.Class对象在任何地方没有被引用，也无法通过反射访问该类的方法。

> 在大量使用反射、动态代理、CGLib等bytecode框架的场景，以及动态生成JSP和OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

3.2 回收收集算法

3.2.1 标记-清除算法（Mark-Sweep）

1. 概念

最基本的算法，主要分为标记和清除2个阶段。首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象

2. 缺点

1. 效率不高，标记和清除过程的效率都不高
2. 空间碎片，会产生大量不连续的内存碎片，会导致大对象可能无法分配，提前触发GC 。

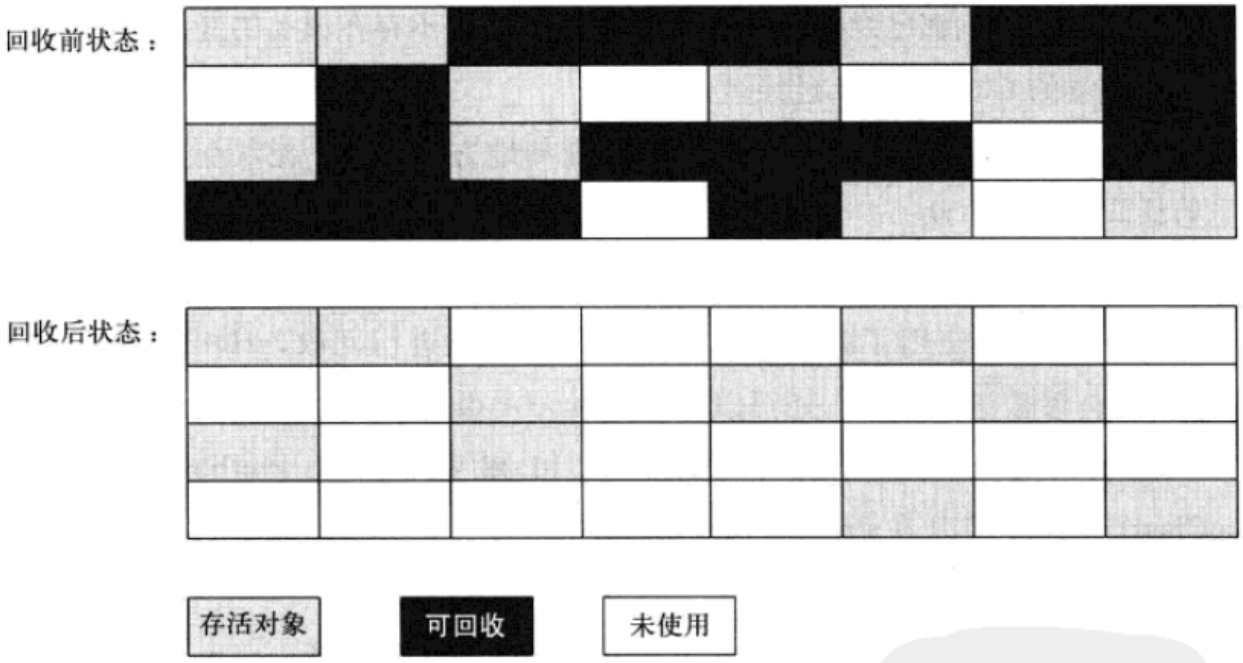


图 3-2 “标记－清除”算法示意图

3.2.2 复制回收算法（Copying）

1. 概念

为解决效率。它将可用内存按容量划分为相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

2. 回收新生代

现在商业虚拟机都是采用这种收集算法来回收新生代，当回收时，将Eden和Survivor中还存活着的对象拷贝到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor的空间。

HotSpot虚拟机默认Eden和Survivor的大小比例是8：1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存是会被“浪费”的。当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

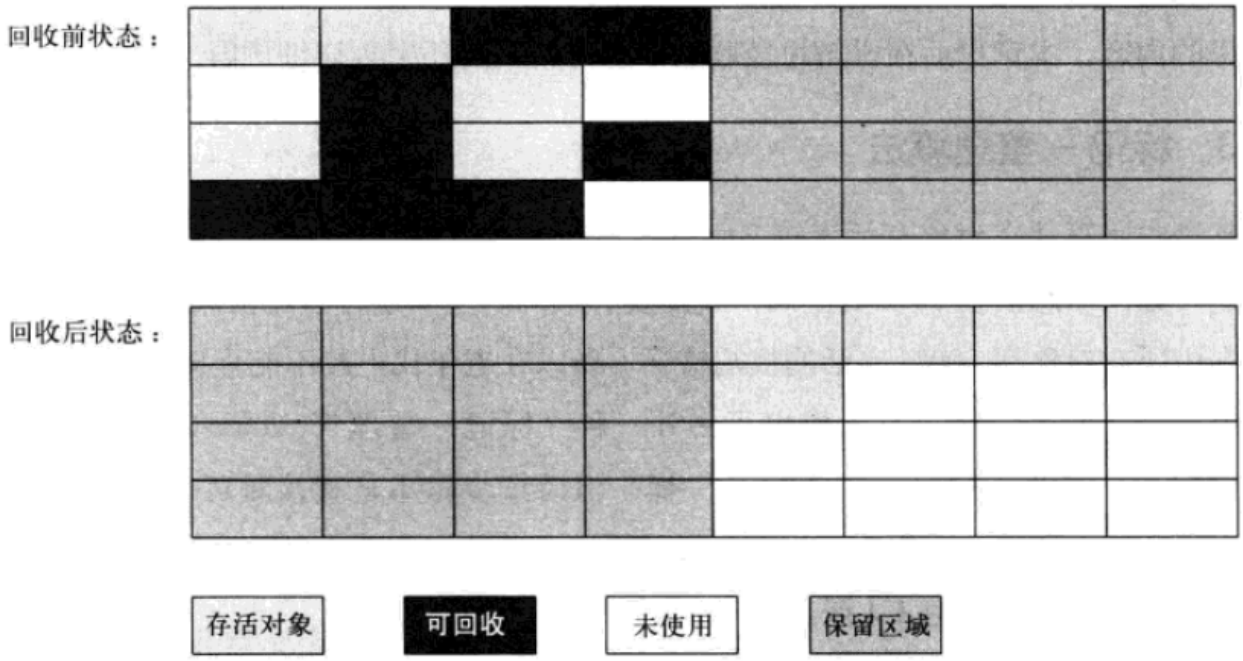


图 3-3　复制算法示意图

3.2.3 标记-整理算法（Mark-Compact）

老年代没有人担保，不能用复制回收算法。可以用标记-整理算法，标记过程仍然与“标记-清除”算法一样，然后让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

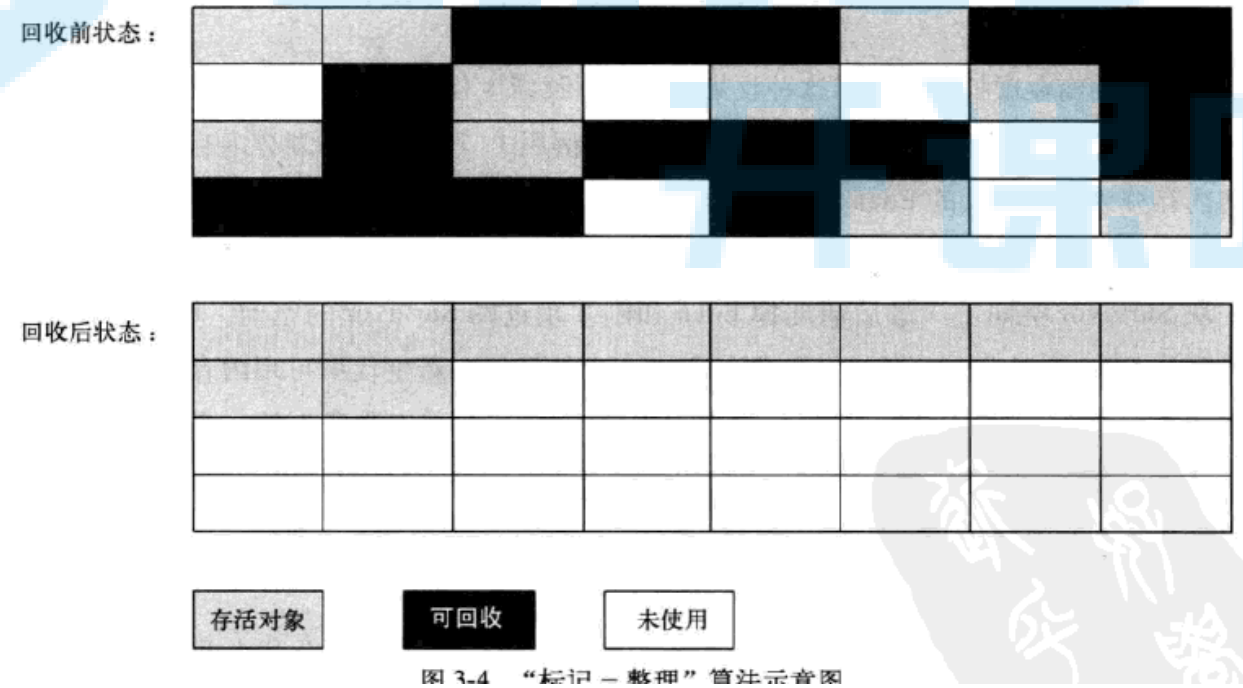


图 3-4　“标记－整理”算法示意图

3.2.4 分代回收算法（Generational Collection）

当前商业虚拟机都是采用这种算法。根据对象的存活周期的不同将内存划分为几块。

- 新生代，每次垃圾回收都有大量对象失去，选择复制算法。
- 老年代，对象存活率高，无人进行分配担保，就必须采用标记清除或者标记整理算法