5分掌握Java代码在JVM中运行的过程

原创 龙虾编程 龙虾编程 2024年12月05日 08:00 浙江

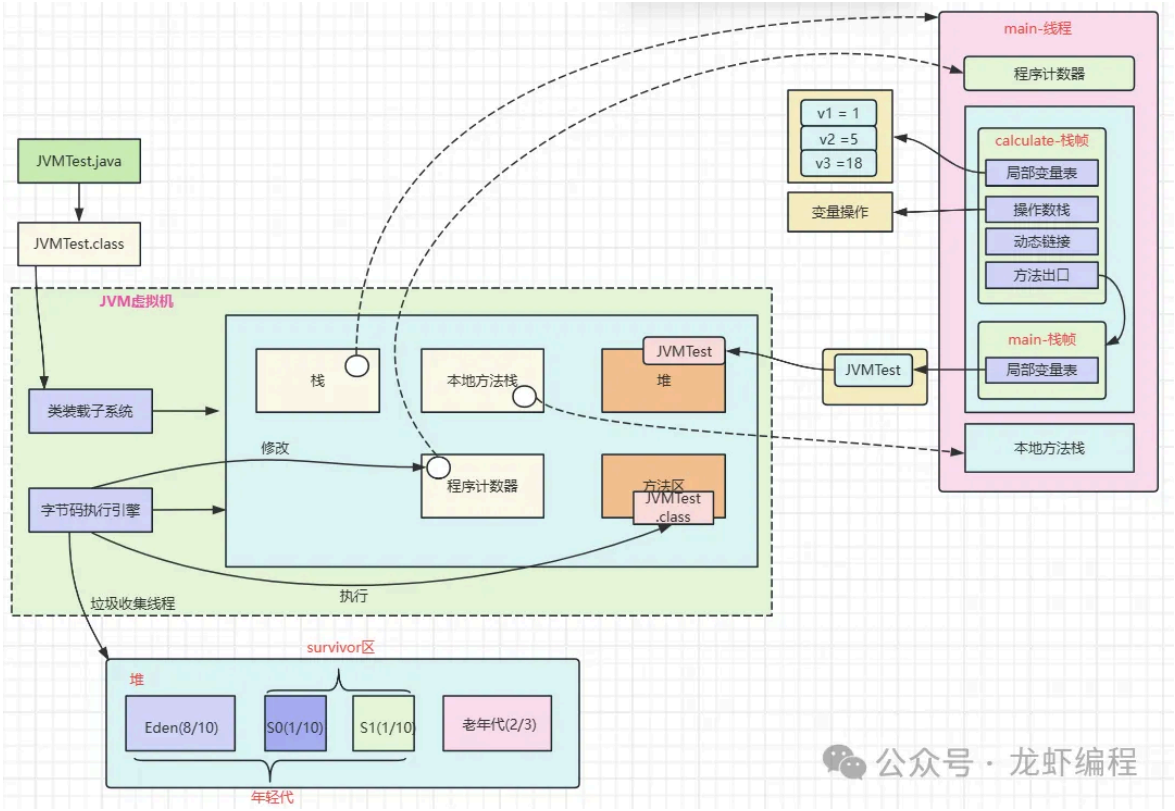
Java中的所有类都必须被装载到JVM中才能运行，JVM中的类加载器负责将编译后的.class文件装载到JVM中，由于.class文件只能在虚拟机上运行，不能直接和操作系统交互，所以JVM需要将文件解释给操作系统，这样就实现和操作系统之间的交互。有如下的代码：

```
public class JVMTest {

    public static void main(String[] args) {
        JVMTest jvmTest = new JVMTest();
        //调用内部方法
        int result = jvmTest.calculate();
        System.out.println(result + "---------------jvmTest end ------------
        //开启一个子线程
        new Thread(()-> System.out.println("我是子线程")).
                start();
    }

    private int calculate() {
        int v1 = 1;
        int v2 = 5;
        int v3 = (v1 + v2) * 5;
        return v3;
    }
}
```

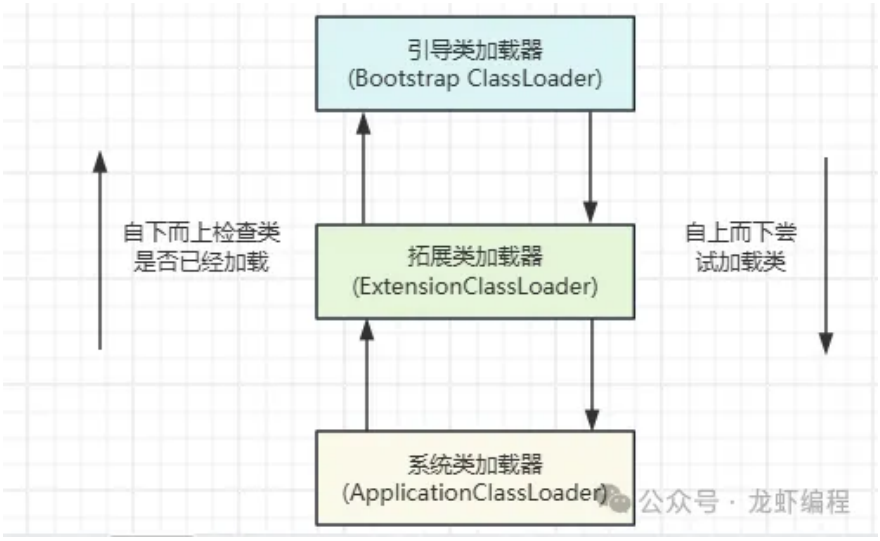
下面我们聊聊上面的代码在JVM中运行的过程，如下是JVM中各个部分的协调工作图：



1、类加载

JVMTest.java文件经过编译之后为JVMTest.class，然后在我们的本地就可以看到一个如下的文件：

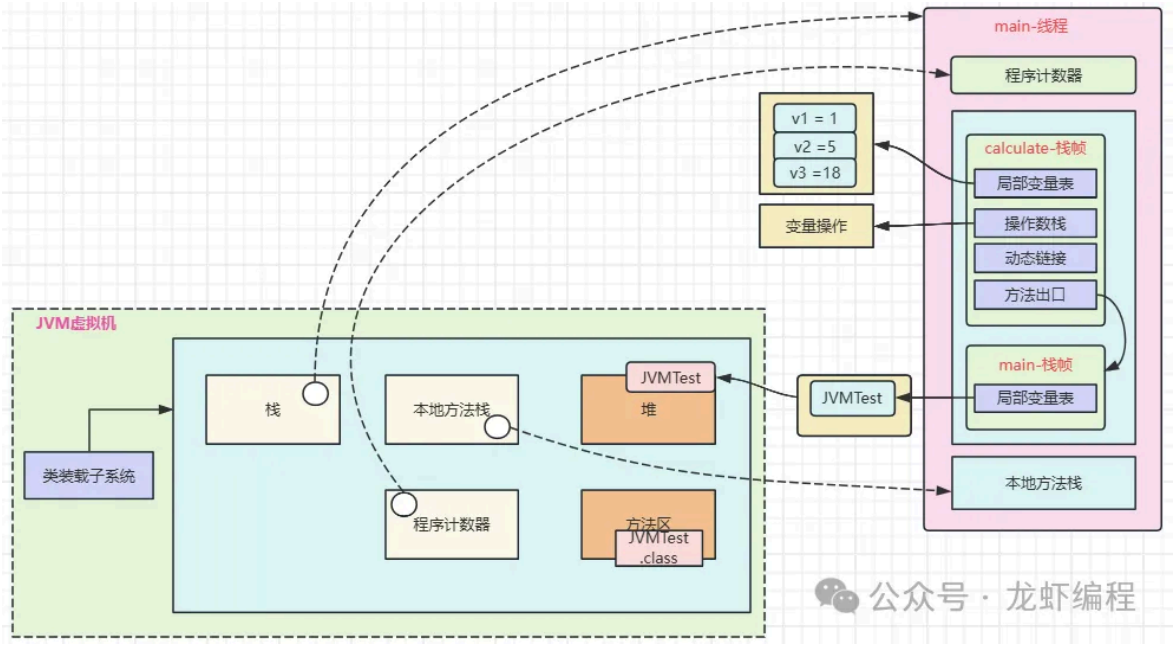
编译后的class文件通过JVM中的类装载子系统装载到JVM中，装载是通过双亲委派机制来实现，其过程如下所示：



双亲委派机制保证类加载的安全性，无论哪个类被加载都会首先让引导类加载器加载，当引导类加载器无法加载这个类的时候才会让子加载器加载，这样避免了类重复加载和类冲突的问题。

在双亲委派机制中子加载器可以使用父加载器加载的类，而父加载器不能使用子加载器加载的类。有时候我们需要打破双亲委派机制，典型有com.mysql.jdbc.Driver类，打破双亲委派机制也很简单，只需要自定义类加载器，重写loadClass()方法即可实现。

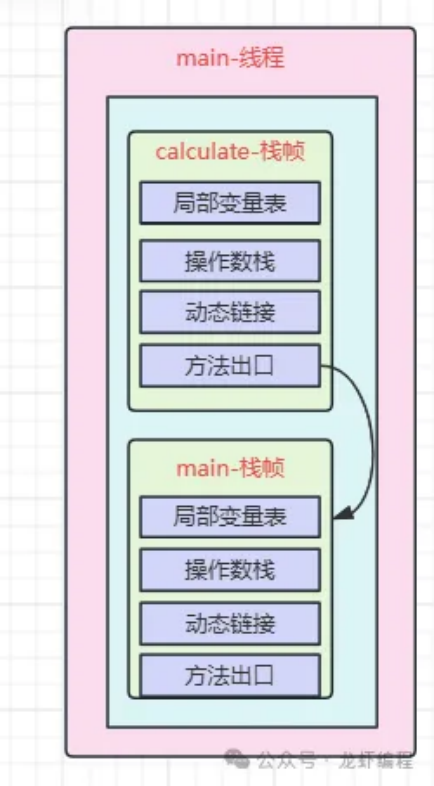
2、代码在JVM中的执行流程



在JVM运行时数据区中，堆和方法区是线程共享的，栈、本地方法栈、程序计数器是线程私有的。

2.1 栈

当执行main方法的时候，在栈上开辟一块空间来存储线程私有栈，由于首先的执行的是main方法，所以首先会在线程栈中给main方法开辟一个方法的栈帧，然后执行calculate方法的时候又开辟一个新的栈帧，栈帧是先进后出（先执行main方法，再执行calculate方法，那么main方法在栈底），如下如所示：



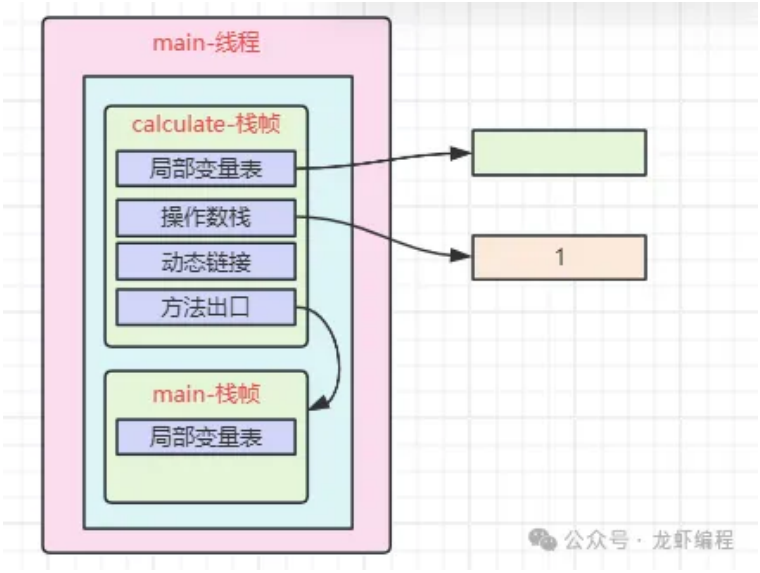
栈帧的设计也是符合实际的执行执行流程，因为先执行完成是后面进入栈的方法，所以这个方法肯定是先被弹出栈的。

每个栈帧中都有局部变量表、操作数栈、动态链接和方法出口等，栈帧中各个部分的功能如下如是：

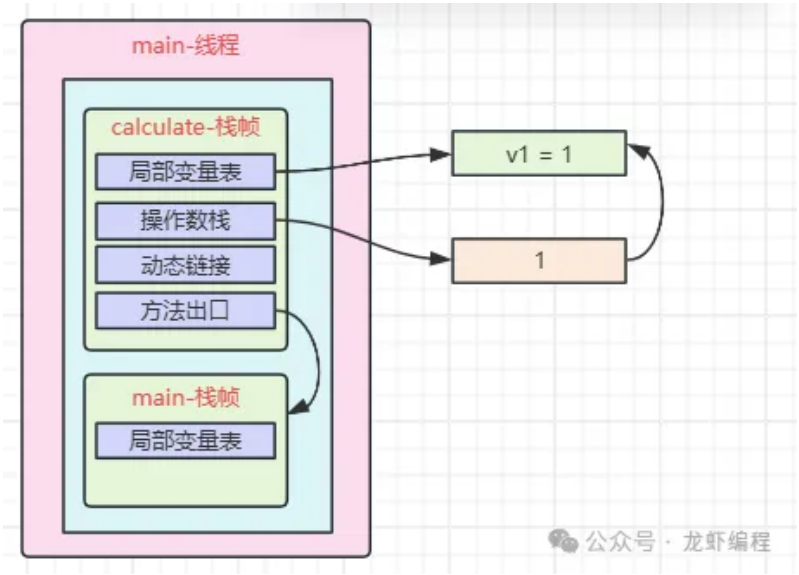
（1）操作数栈和局部变量表

```
17    private int calculate() {
18        int v1 = 1;
19        int v2 = 5;
20        int v3 = (v1 + v2) * 5;
21        return v3;
22    }
```

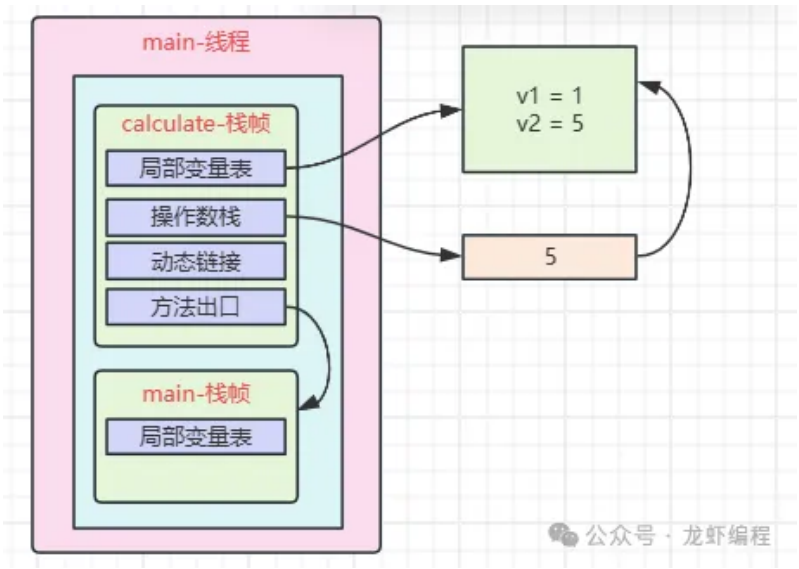

当执行v1 = 1的时，首先在操作数栈中加载1进来，如下图所示：



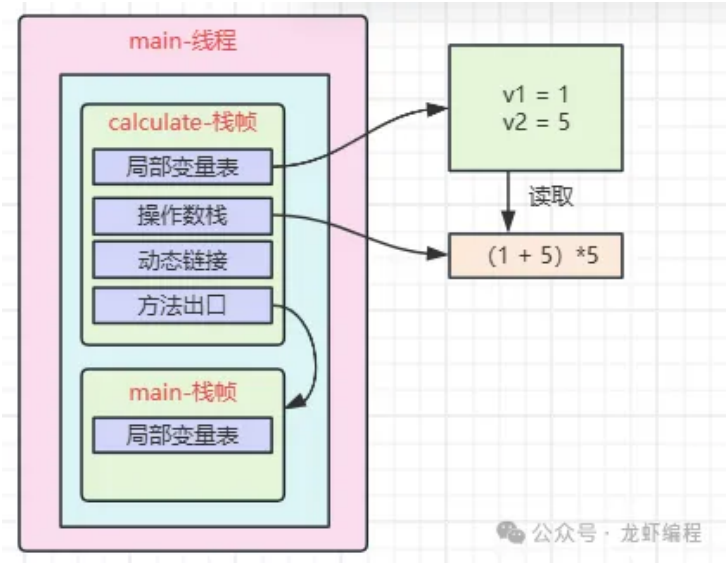
然后在局部变量表中将v1 = 1存储下来，如下所示：



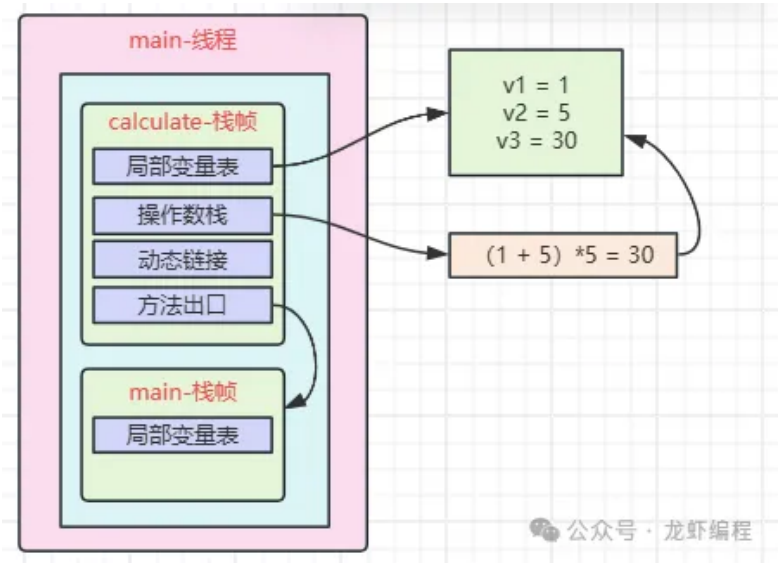
同样的过程加载了v2 = 5到局部变量表中，如下图所示：



当执行（v1 + v2）* 5的时候，先加载v1和v2对应值到操作数栈上，然后执行（1 + 5）* 5，如下所示：



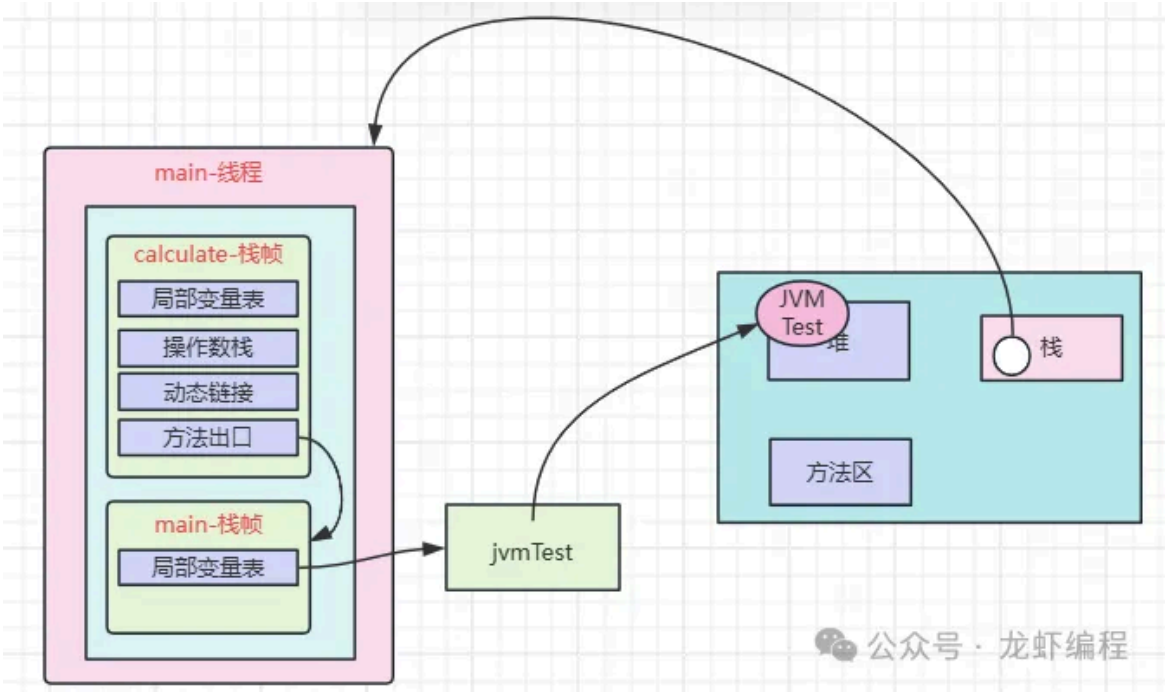
在操作数栈上计算结束之后在将计算的值保存到局部变量表上，如下所示：



当执行如下的代码的时候

```
5  public static void main(String[] args) {
6      JVMTest jvmTest = new JVMTest();
7
```

JVMTest对象是存储在堆中，在栈上的局部变量表中有一个变量引用堆上的对象（需要注意不是所有的对象都是在堆上分配的），如下如图所示：



（2）动态链接

```java
public static void main(String[] args) {
    JVMTest jvmTest = new JVMTest();

    //调用方法
    int result = jvmTest.calculate();
    System.out.println(result + "----------------jvmTest end --------------------");

    //开启一个子线程
    new Thread(()-> System.out.println("我是子线程")).
            start();
}

private int calculate() {
    int v1 = 1;
    int v2 = 5;
    int v3 = (v1 + v2) * 5;
    return v3;
}
}
```


当执行到calculate方法的时候，由于在编译阶段是使用符号引用来标识的，当程序在JVM中运行的时候，需要找到calculate方法中的实际代码，也就是如下的代码：

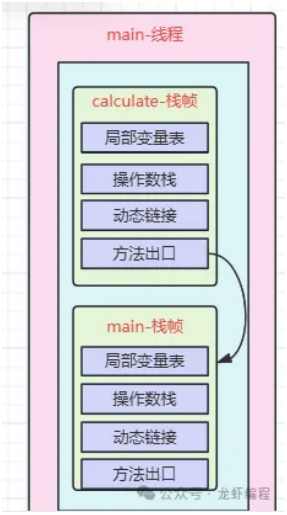
```java
private int calculate() {
    int v1 = 1;
    int v2 = 5;
    int v3 = (v1 + v2) * 5;
    return v3;
}
```


我们知道calculate方法中的代码编译之后被JVM类加载器加载到方法区中，那么现在只需要找到方法区calculate方法就可以找到calculate方法中需要被执行的代码，此

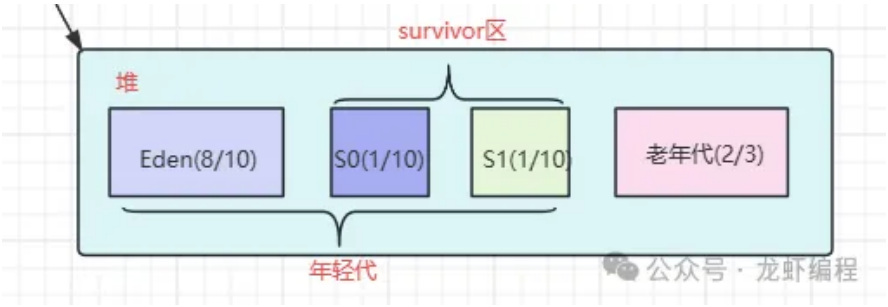
时方法区上calculate方法的入口地址我们称之为直接引用，我们将这个calculate方法的入口地址放在动态链接中。

（3）方法出口

当calculate方法执行完成之后需要回到main方法中继续执行代码，那么就要记录calculate方法结束后要回到main方法的哪个位置继续执行，方法出口中就是记录这个位置，如下图所示：

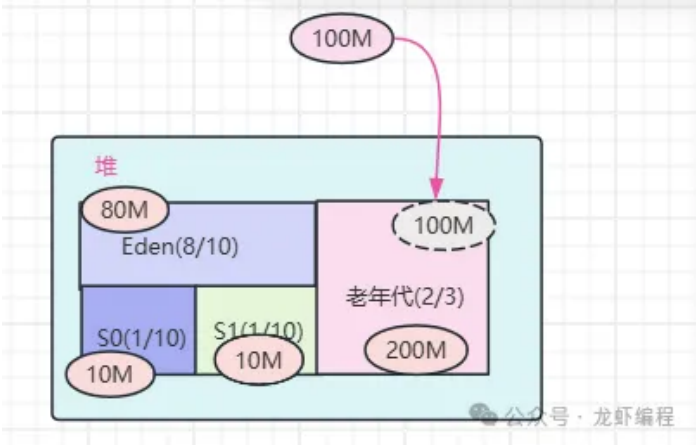


2.2 堆



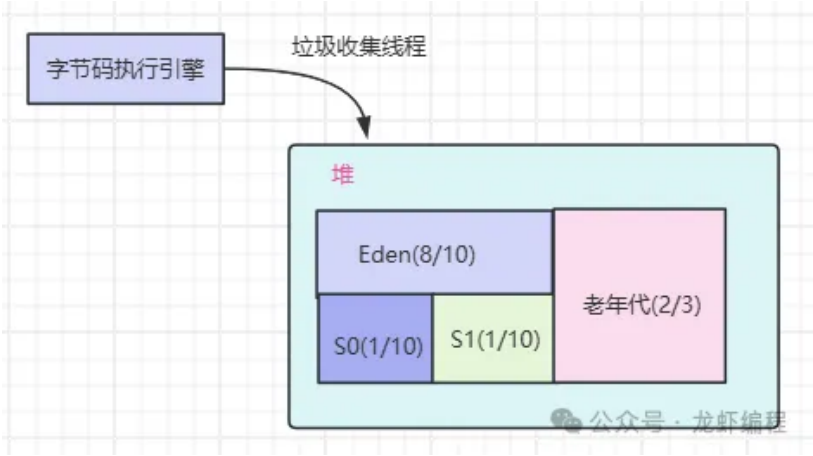
堆是所有线程共享的区域，它由年轻代和老年代组成，新创建的对象会存在放在Eden区，当Eden无空间的存放的时候会触发minor gc将依然存活的对象存放到survivor区域（survivor区域的S0和S1中一个区域存放对象，另一个区域是预留的，当一个区域满了之后，会将存活的对象全部移动到另一个区域中），在hotspot虚拟机中，survivor区域中对象在S0和S1来回移动15次（因为对象头中有4位专门用来标识对象的gc年龄，由于4位最大可以表示15，所以是15次）之后的对象如果依然存活就会放入老年代。

如果新创建的对象比较大，年轻代无法存放的时候，会直接存放到老年代上，如下图所示：

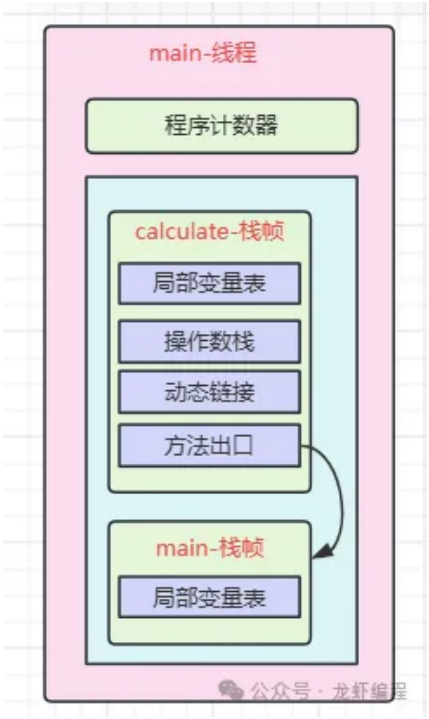


假设Eden区是80M，S0和S1区分别是10M，此时一个新对象是90M，那么对象将会被分配代老年代存储。

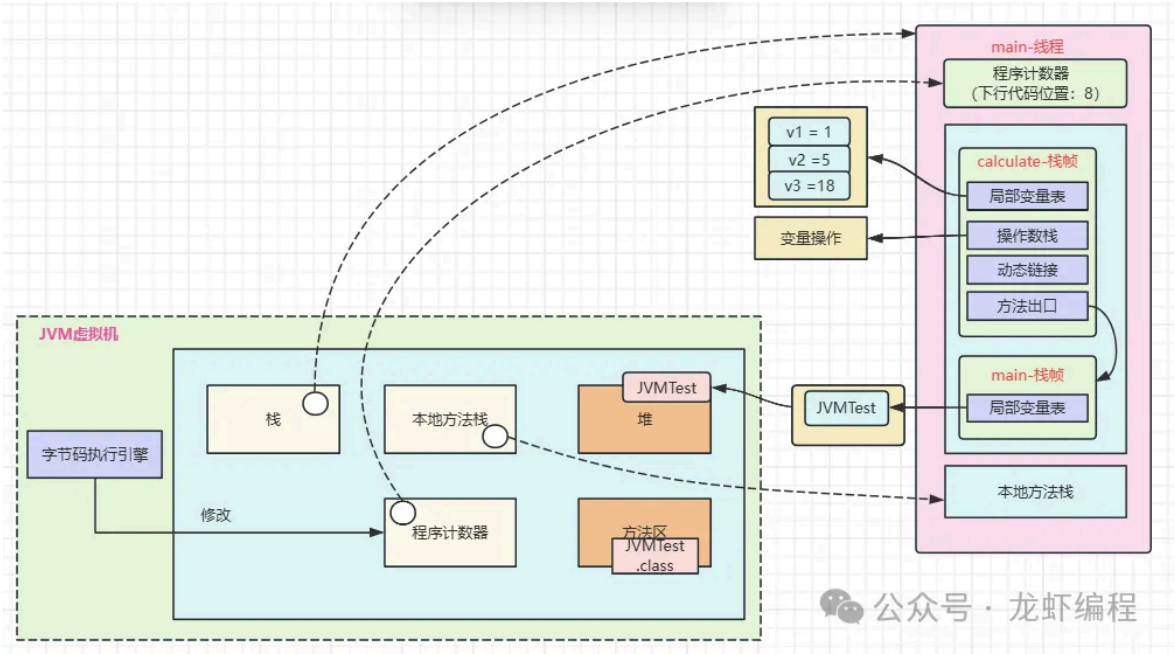
如果老年代空间也不足的时候，就会触发fullGC，fullGC会对整个堆进行垃圾清理，如果还没有清理出足够的空间来存放对象就会OOM。执行垃圾回收操作是字节码执行引擎中的垃圾收集线程，如下所示：



2.3 程序计数器

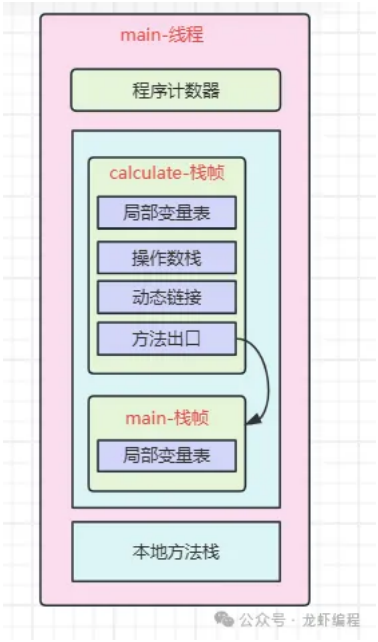


在每个线程栈都有一个线程私有的程序计数器，它用来标记当前线程下一行代码的位置，由于我们CPU是有时间片的，如果时间片用完之后当前的线程就无法再执行，等下一个CPU的时间片到来的时候从程序计数器中拿到线程执行下一行代码的位置继续执行，如下图所示：



程序计数器中的下一行代码执行位置是字节码执行引擎保存的到线程栈中的程序计数器中。

2.4 本地方法栈



本地方法栈储存的是本地接口库里调用的方法，在java里面就是native关键字修饰的方法。native修饰的方法，这些方法执行也是需要内存的（如存放临时变量），所以在本地方法栈上将调用native关键字修饰的方法开辟一块专用的内存空间。

native关键字修饰的方法JVM会去调用底层C/C++语言的库，典型是线程的start方法，如下所示：

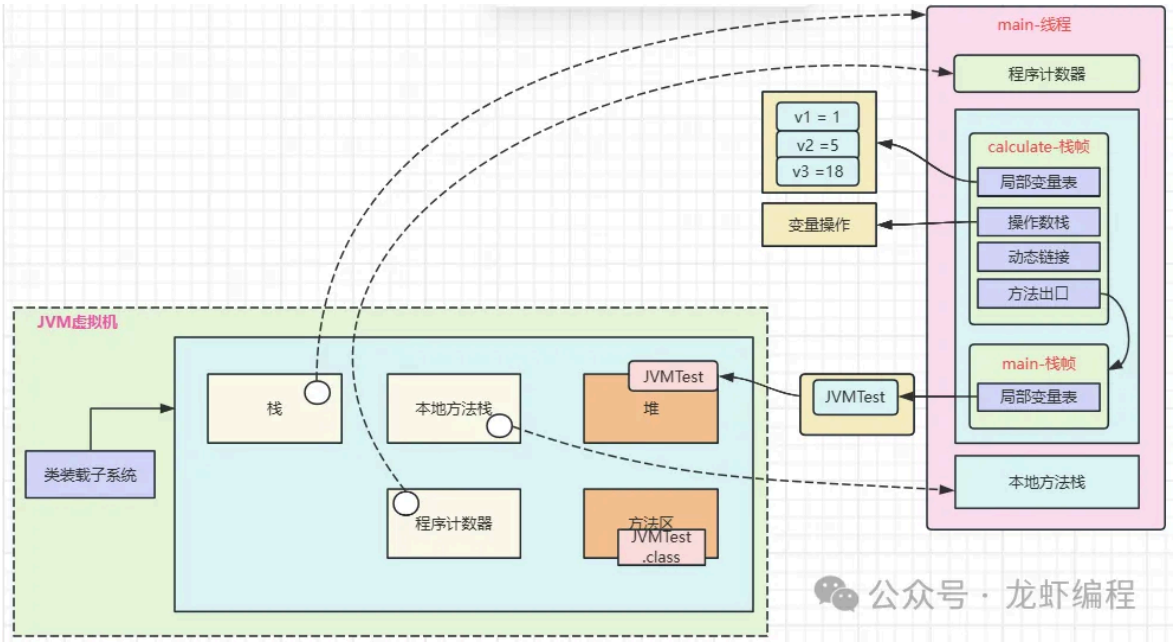
```
public static void main(String[] args) {
    JVMTest jvmTest = new JVMTest();

    //调用方法
    int result = jvmTest.calculate();
    System.out.println(result + "---------------jvmTest end --------------------");

    //开启一个子线程
    new Thread(()-> System.out.println("我是子线程")).
            start();
}
```

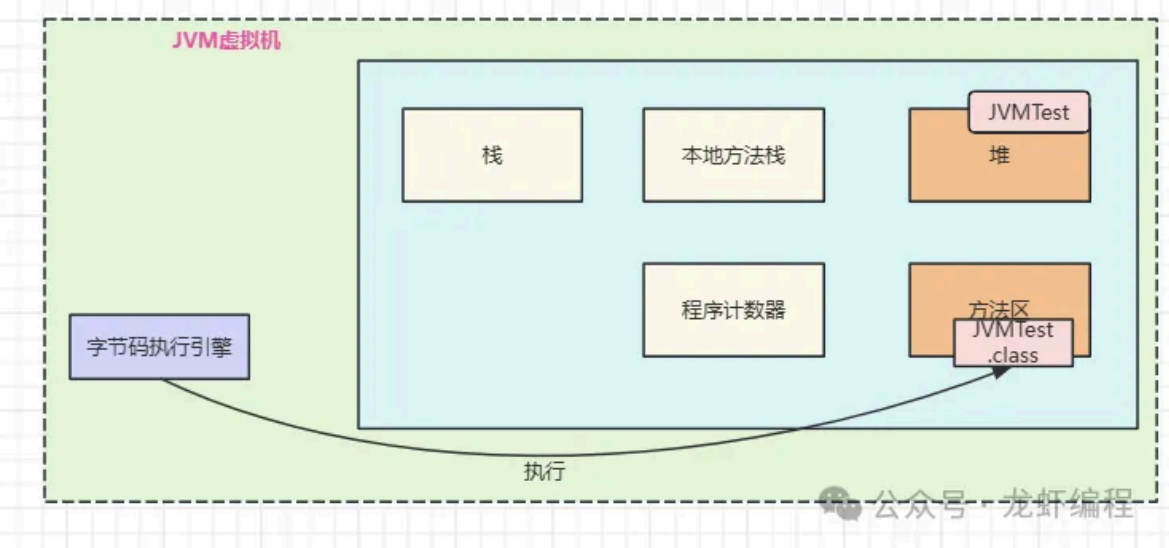
```
713         group.add(this);
714
715         boolean started = false;
716         try {
717             start0();
718             started = true;
719         } finally {
720             try {
721                 if (!started) {
722                     group.threadStartFailed( t: this);
723                 }
724             } catch (Throwable ignore) {
725                 /* do nothing. If start0 threw a Throwable then
726                   it will be passed up the call stack */
727             }
728         }
729     }
730
731     private native void start0();
```


native方法执行中会在本地方法栈上开辟一块空间，如下图所示：

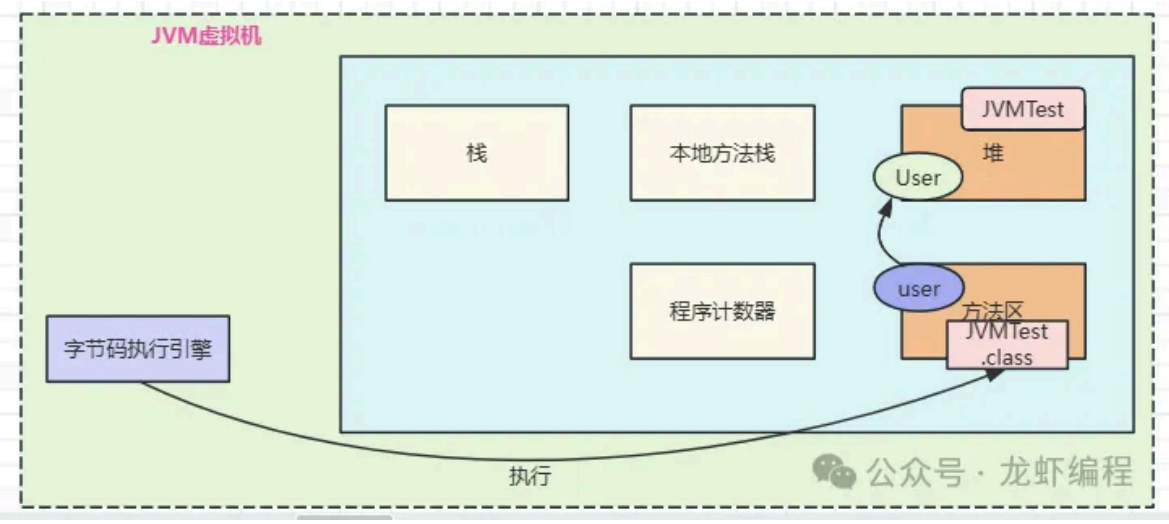


2.5 方法区

方法区存储是线程共享的区域，方法区中主要存放常量池、静态变量(static)以及类信息等，jdk8之后方法区使用的是堆外内存。



方法区中编译的类文件是由字节码执行引擎执行的，它和堆也存在联系，如果类中存在静态的对象，那么对象的变量会存在到方法区，通过指针指向堆上的对象，如下所示：



总结：

（1）程序开发人员开发的代码会先被编译成class文件，然后class文件被类加载器加载到JVM中

（2）JVM中运行时数据区主要有栈、堆、方法区、程序计数器和本地方法组成，通过它们之间的配合完成代码的执行。

（3）字节码执行引擎执行方法的class文件，修改程序计数器的下一行代码执行位置和开启垃圾收集线程进行垃圾回收。

面试 49

面试 · 目录

上一篇 下一篇

快速掌握分布式一致性算法Paxos

敏感数据加密后如何实现模糊查询