实战Java虚拟机-基础篇

Java内存区域

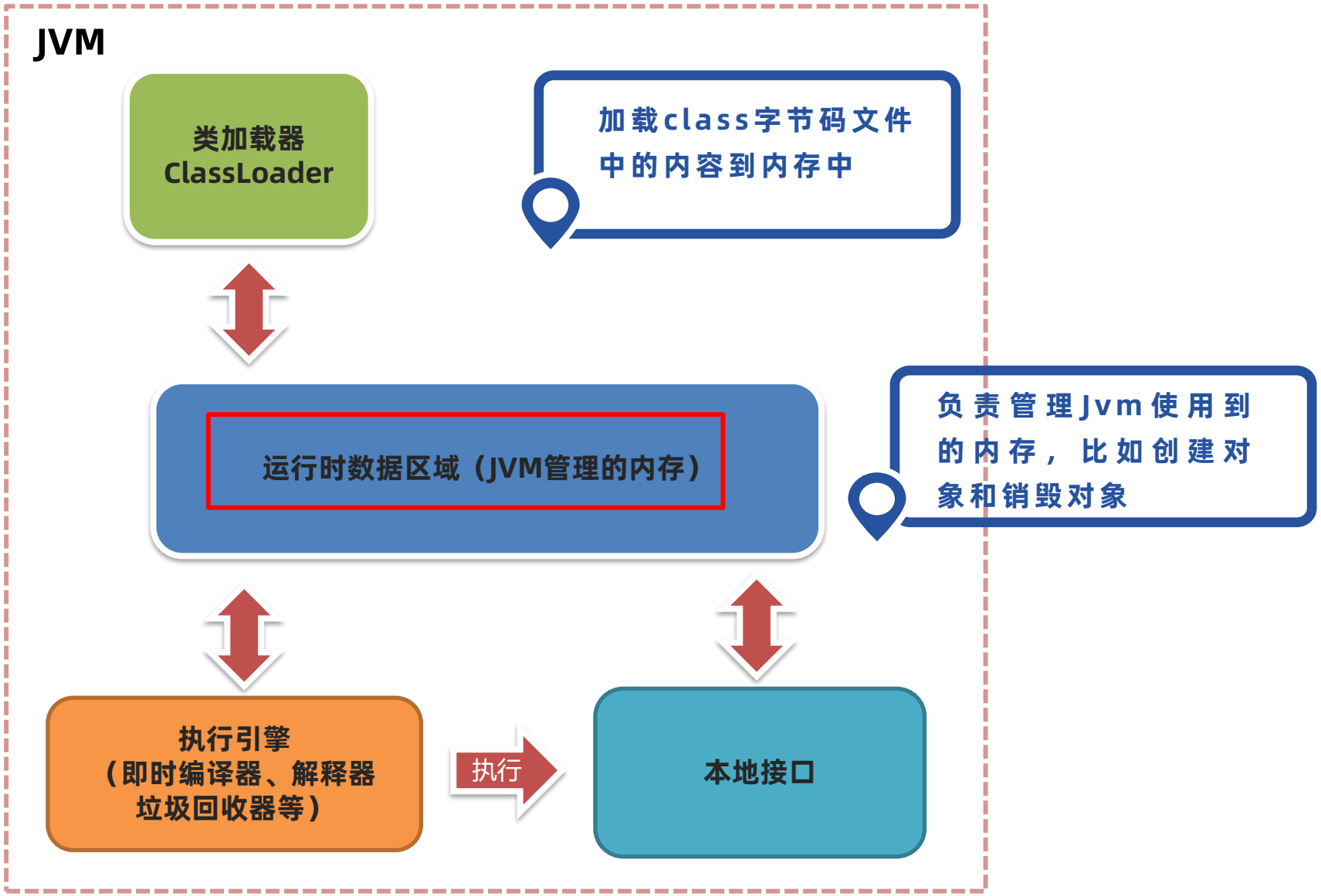
黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

JVM的组成

【字节码文件】

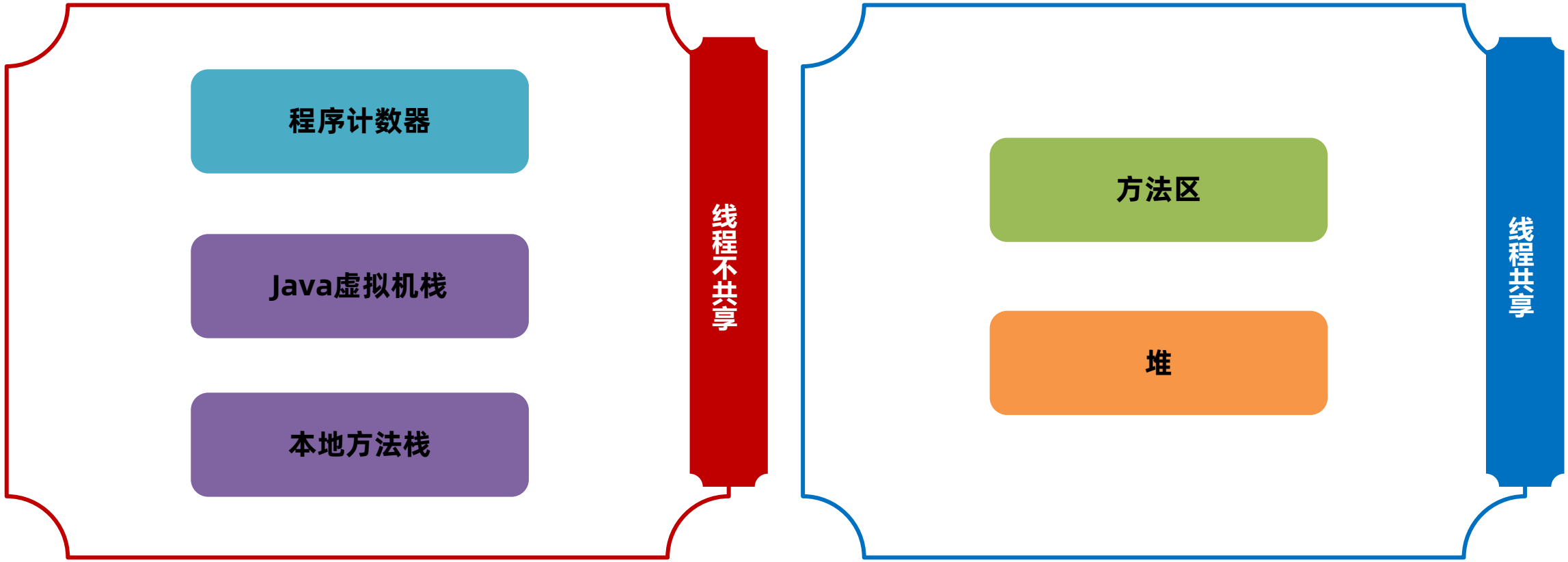
加载



03 运行时数据区

运行时数据区-总览

- Java虚拟机在运行Java程序过程中管理的内存区域，称之为运行时数据区。
- 《Java虚拟机规范》中规定了每一部分的作用。



运行时数据区 – 应用场景

面试官

Java的内存分成哪几部分？详细介绍一下吧

Java内存中哪些部分会内存溢出？

JDK7和8中在内存结构上的区别是什么？

解决面试难题

运行时数据区 – 应用场景

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2722843	root	20	0	4224656	1.3g	17972	S	82.1	69.3	0:07.75	java

```
java.lang.OutOfMemoryError: Java heap space
        at com.itheima.jvmoptimize.entity.UserEntity.<init>(UserEntity.java:11) ~[classes!/:0.0.1-SNAPSHOT]
        at com.itheima.jvmoptimize.controller.OOMController.createCache(OOMController.java:34) ~[classes!/:0.0.1-SNAPSHOT]
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_372]
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0_372]
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0_372]
        at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0_372]
        at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:205) ~[spring-web-5.3.27.jar!/
:5.3.27]
        at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:150) ~[spring-web-5.3.
27.jar!/:5.3.27]
        at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod
.java:117) ~[spring-webmvc-5.3.27.jar!/:5.3.27]
        at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapt
```

工作中的实际问题 - 内存溢出

运行时数据区 – 应用场景

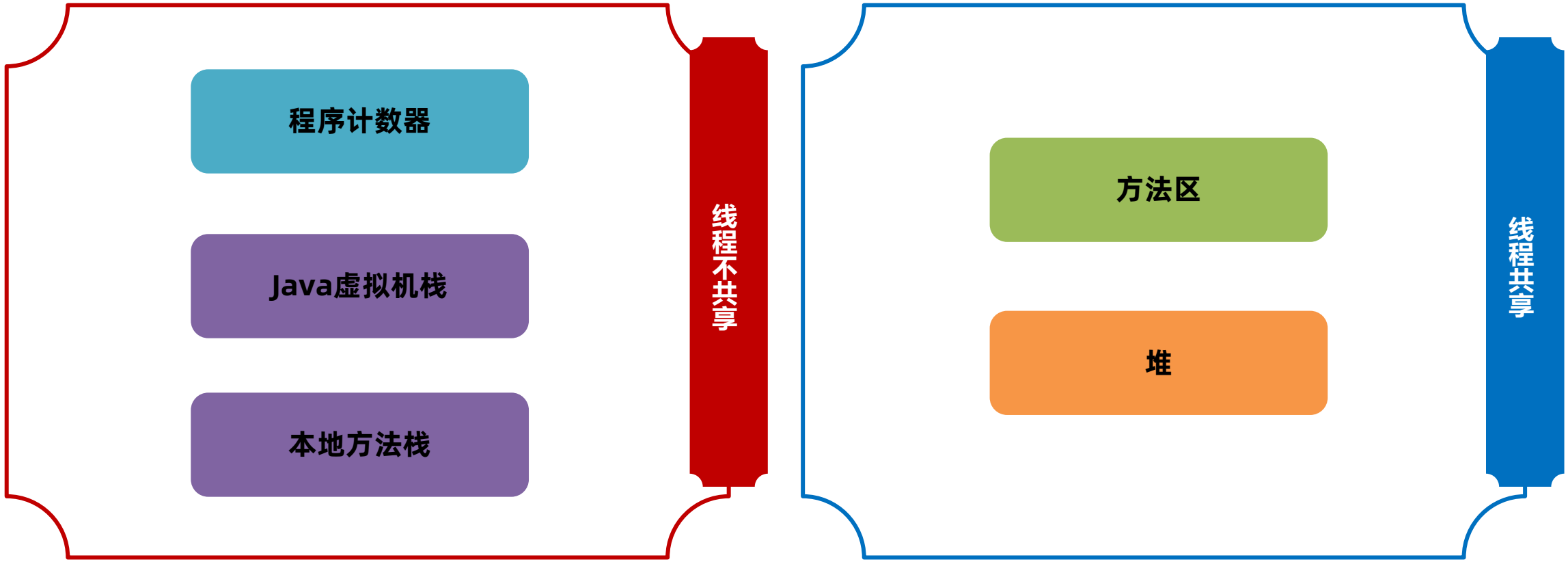
内存调优学习路线

目录

Contents

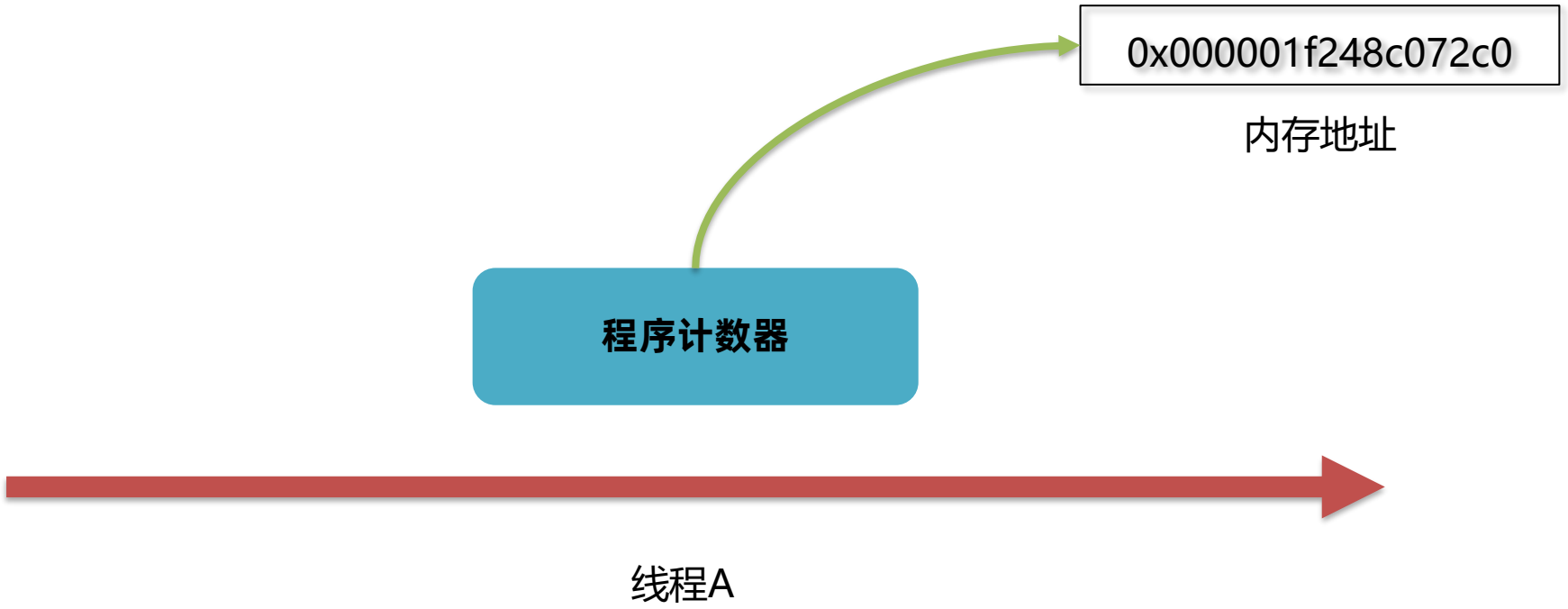
运行时数据区-总览

- Java虚拟机在运行Java程序过程中管理的内存区域，称之为运行时数据区。
- 《Java虚拟机规范》中规定了每一部分的作用。



程序计数器

- 程序计数器（**Program Counter Register**）也叫**PC**寄存器，每个线程会通过程序计数器记录当前要执行的的字节码指令的地址。



程序计数器

- 一个程序计数器的具体案例：

```
public static void main(String[] args) {
    int i = 0;
    if(i == 0){
        i--;
    }
    i++;
}
```

源代码

编译

```
0 iconst_0
1 istore_1
2 iload_1
3 ifne 9 (+6)
6 iinc 1 by -1
9 iinc 1 by 1
12 return
```

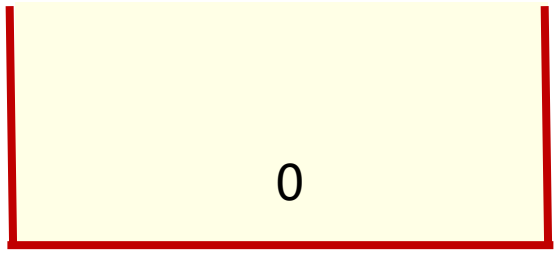
将局部变量i赋值为0

判断i和0如果不相等
跳转到指令位置9

i和0相等，就将i减1

i和0不相等，就将i加1

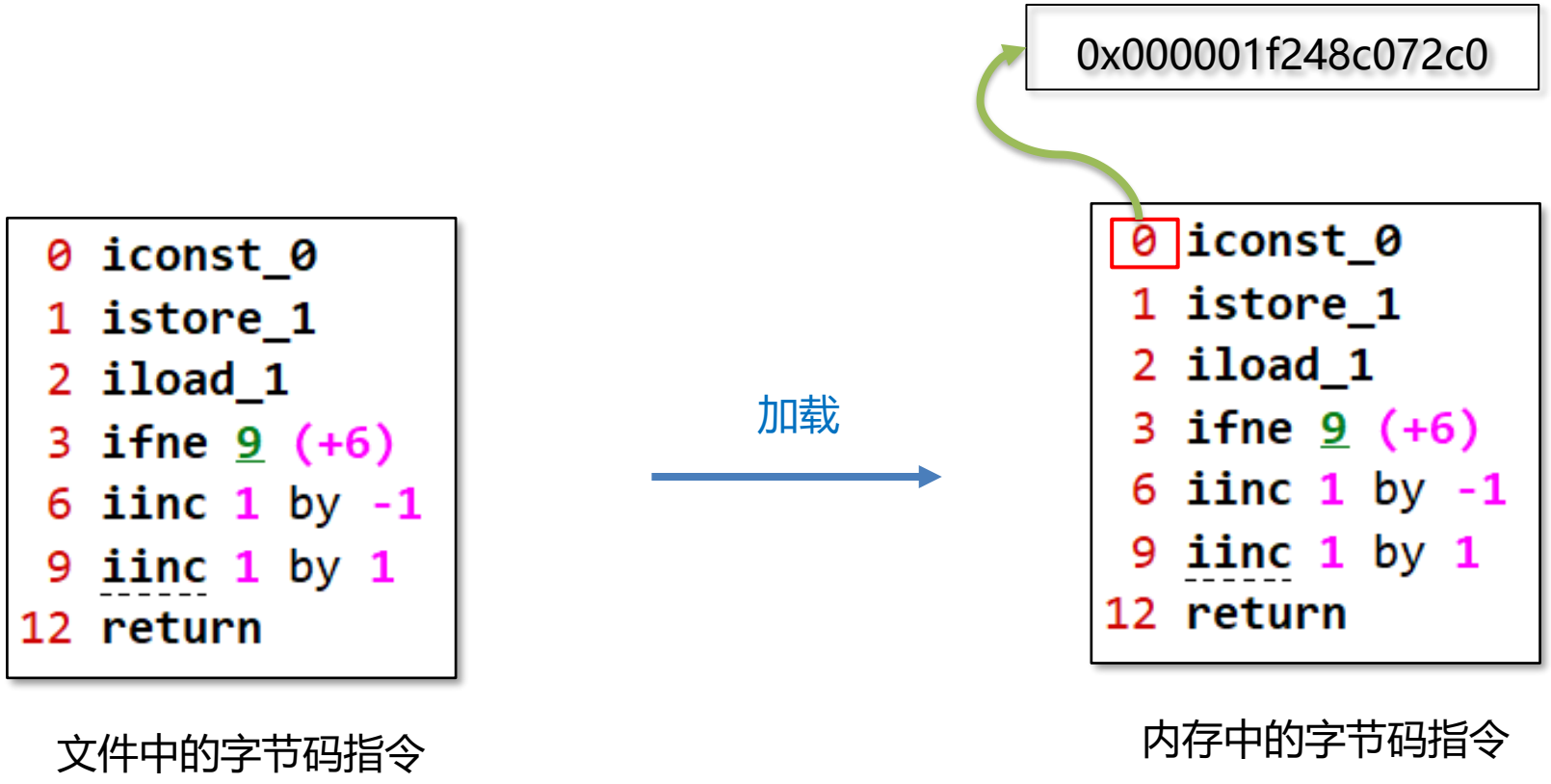
字节码指令



操作数栈

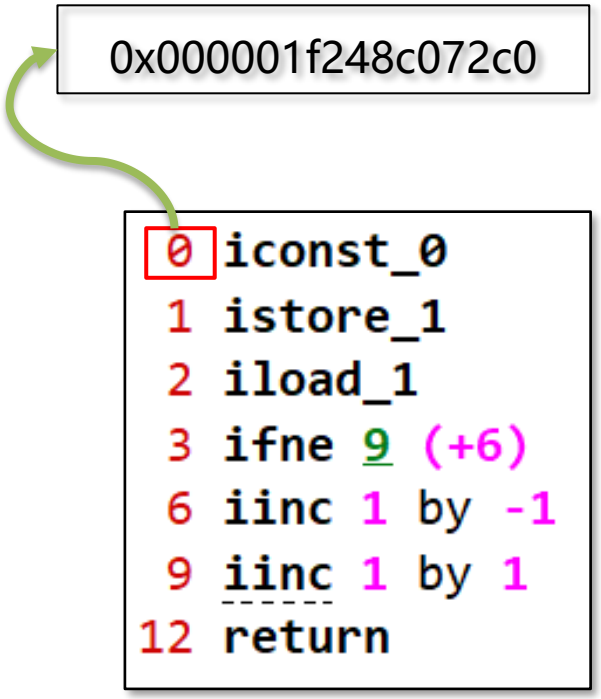
程序计数器

- 在加载阶段，虚拟机将字节码文件中的指令读取到内存之后，会将原文件中的偏移量转换成内存地址。每一条字节码指令都会拥有一个内存地址。

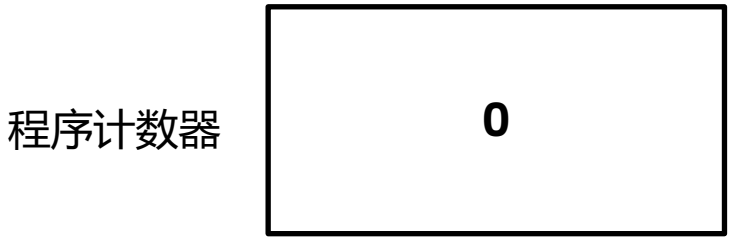


程序计数器

- 在代码执行过程中，程序计数器会记录下一行字节码指令的地址。执行完当前指令之后，虚拟机的执行引擎根据程序计数器执行下一行指令。



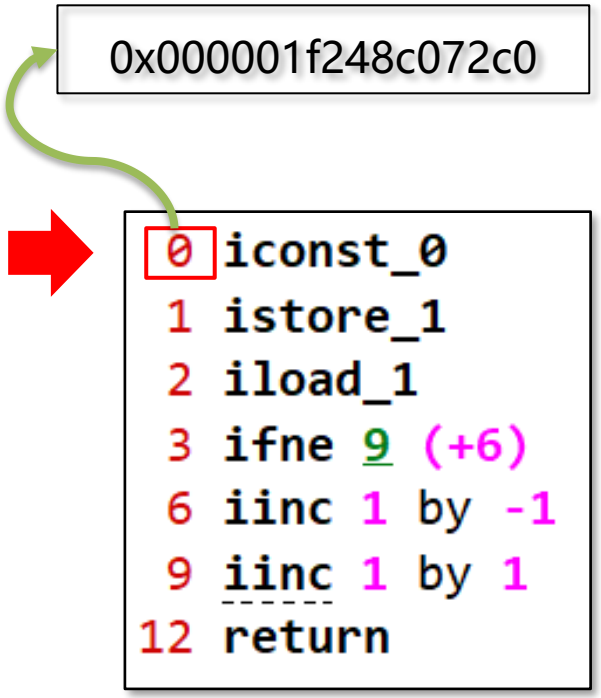
内存中的字节码指令



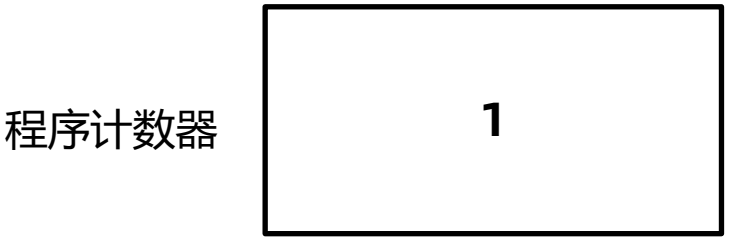
为了更直观展示，用偏移量来替代

程序计数器

- 在代码执行过程中，程序计数器会记录下一行字节码指令的地址。执行完当前指令之后，虚拟机的执行引擎根据程序计数器执行下一行指令。



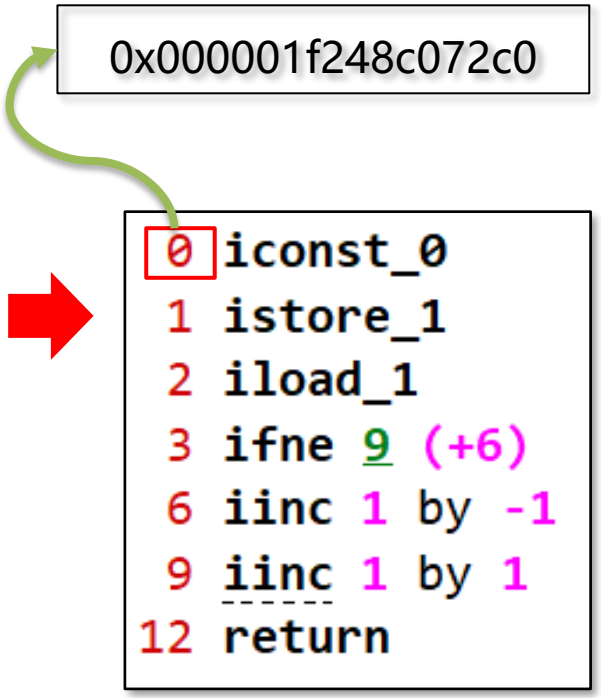
内存中的字节码指令



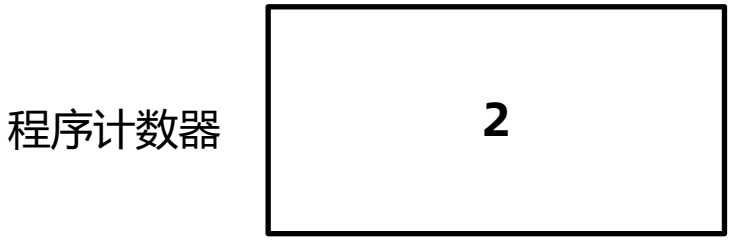
为了更直观展示，用偏移量来替代

程序计数器

- 在代码执行过程中，程序计数器会记录下一行字节码指令的地址。执行完当前指令之后，虚拟机的执行引擎根据程序计数器执行下一行指令。



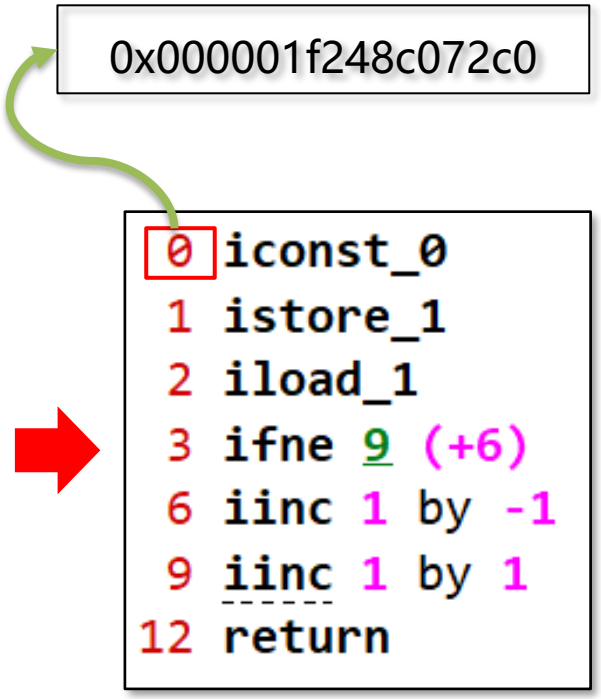
内存中的字节码指令



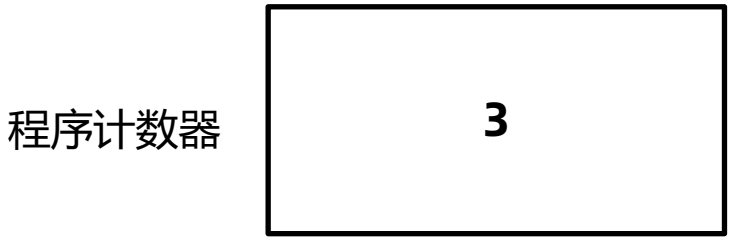
为了更直观展示，用偏移量来替代

程序计数器

- 在代码执行过程中，程序计数器会记录下一行字节码指令的地址。执行完当前指令之后，虚拟机的执行引擎根据程序计数器执行下一行指令。



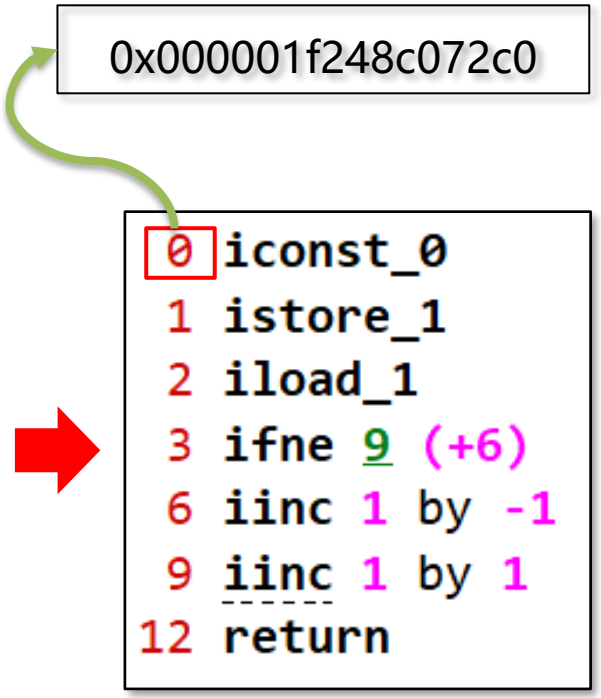
内存中的字节码指令



为了更直观展示，用偏移量来替代

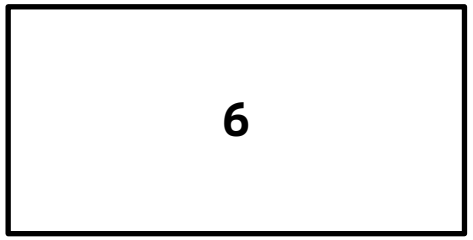
程序计数器

- 在代码执行过程中，程序计数器会记录下一行字节码指令的地址。执行完当前指令之后，虚拟机的执行引擎根据程序计数器执行下一行指令。



内存中的字节码指令

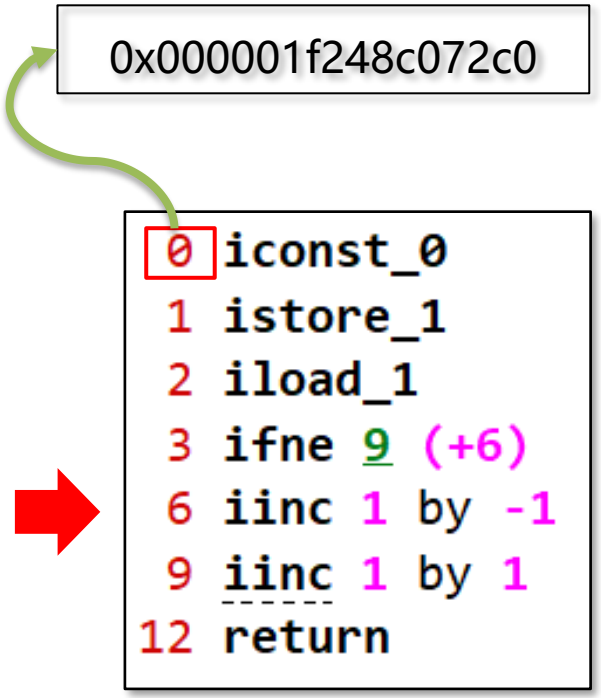
程序计数器



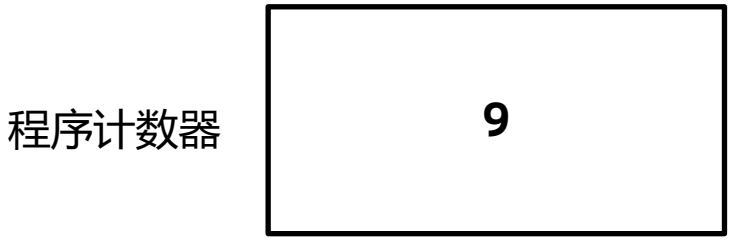
为了更直观展示，用偏移量来替代

程序计数器

- 在代码执行过程中，程序计数器会记录下一行字节码指令的地址。执行完当前指令之后，虚拟机的执行引擎根据程序计数器执行下一行指令。



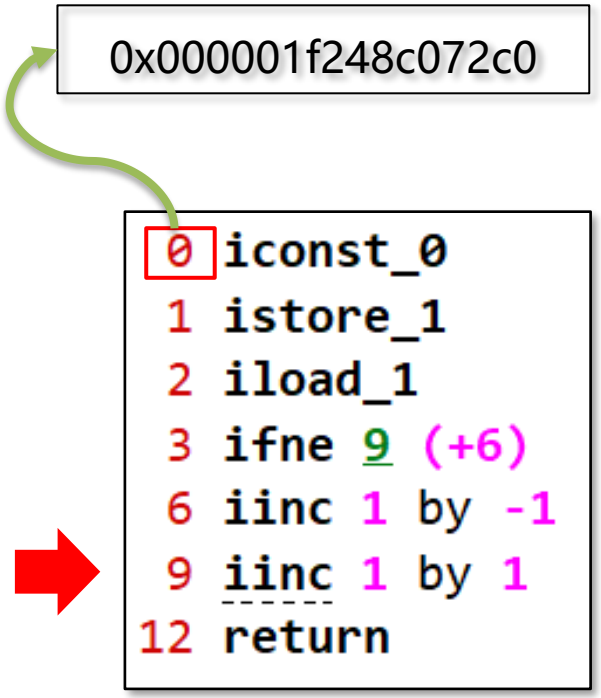
内存中的字节码指令



为了更直观展示，用偏移量来替代

程序计数器

- 在代码执行过程中，程序计数器会记录下一行字节码指令的地址。执行完当前指令之后，虚拟机的执行引擎根据程序计数器执行下一行指令。

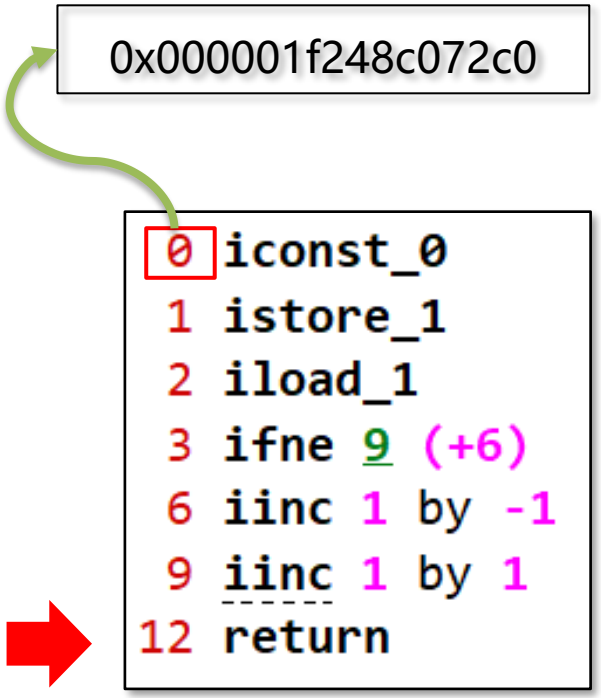


内存中的字节码指令

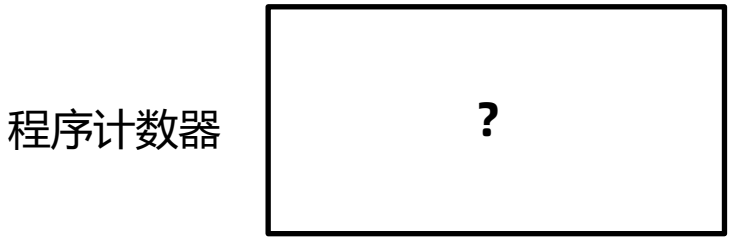
为了更直观展示，用偏移量来替代

程序计数器

- 程序计数器可以控制程序指令的进行，实现分支、跳转、异常等逻辑。



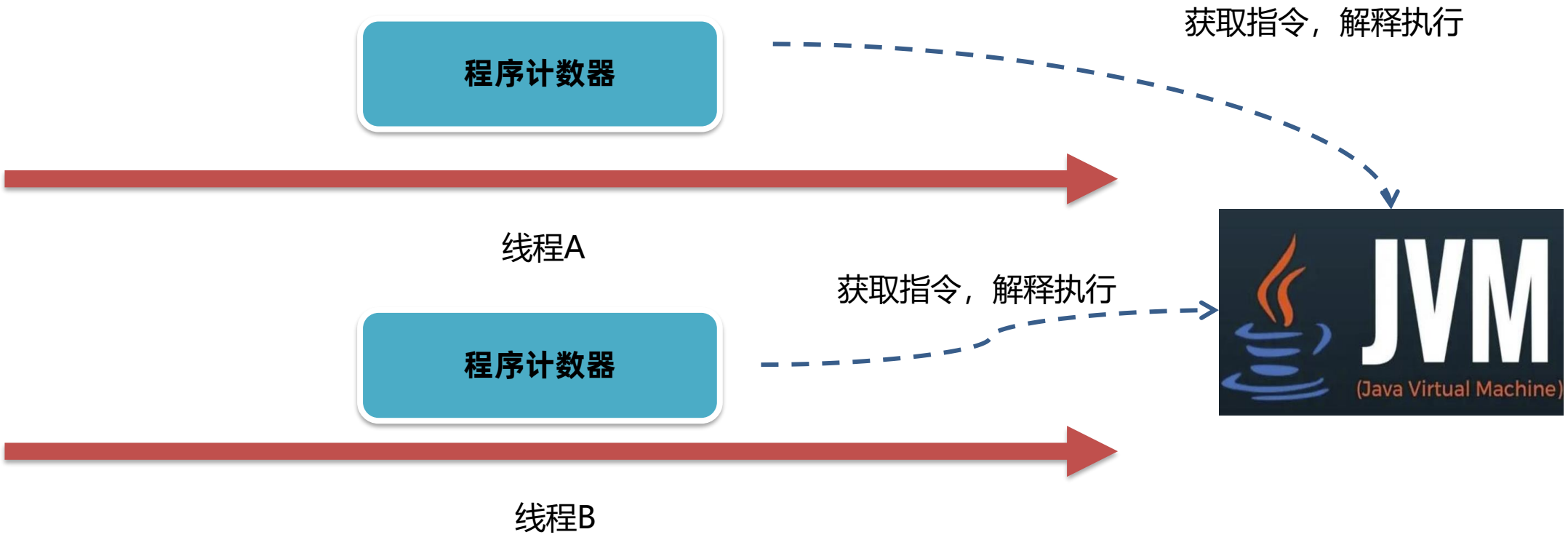
内存中的字节码指令



为了更直观展示，用偏移量来替代

程序计数器

- 在多线程执行情况下，Java虚拟机需要通过程序计数器记录CPU切换前解释执行到那一句指令并继续解释运行。



问题 程序计数器在运行中会出现内存溢出吗？

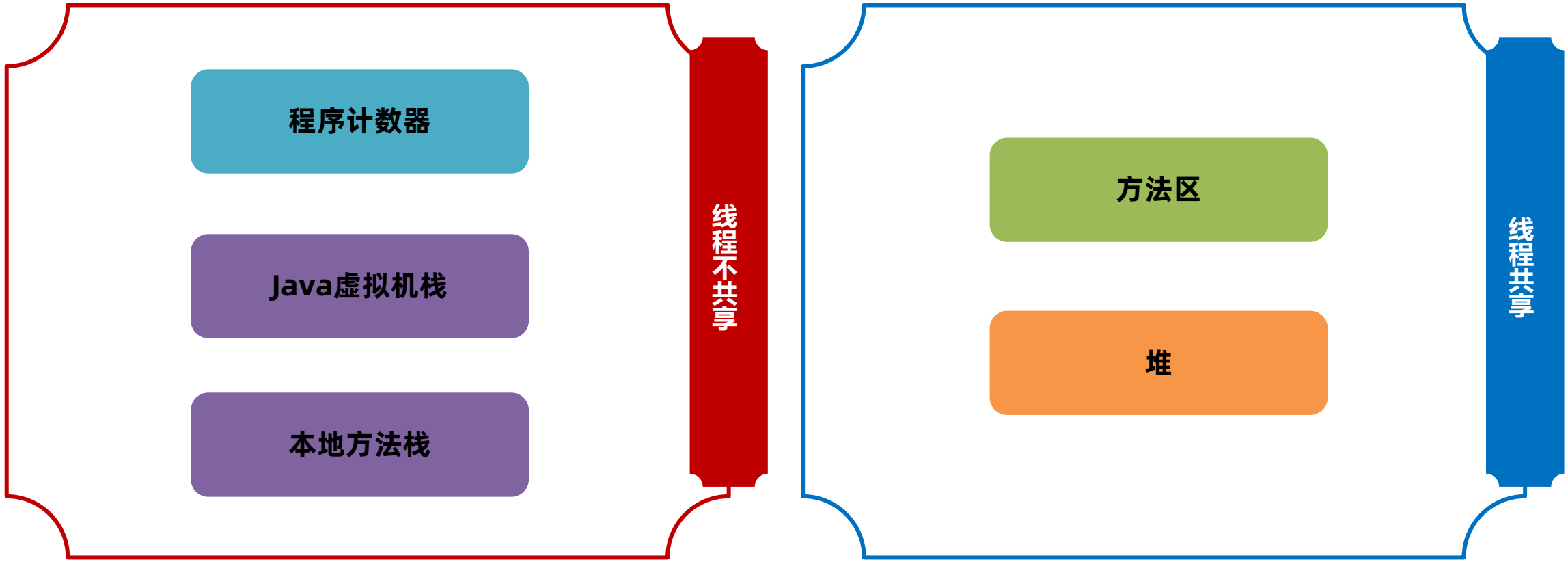
- 内存溢出指的是程序在使用某一块内存区域时，存放的数据需要占用的内存大小超过了虚拟机能提供的内存上限。
- 因为每个线程只存储一个固定长度的内存地址，程序计数器是不会发生内存溢出的。
- 程序员无需对程序计数器做任何处理。

目录

Contents

运行时数据区-总览

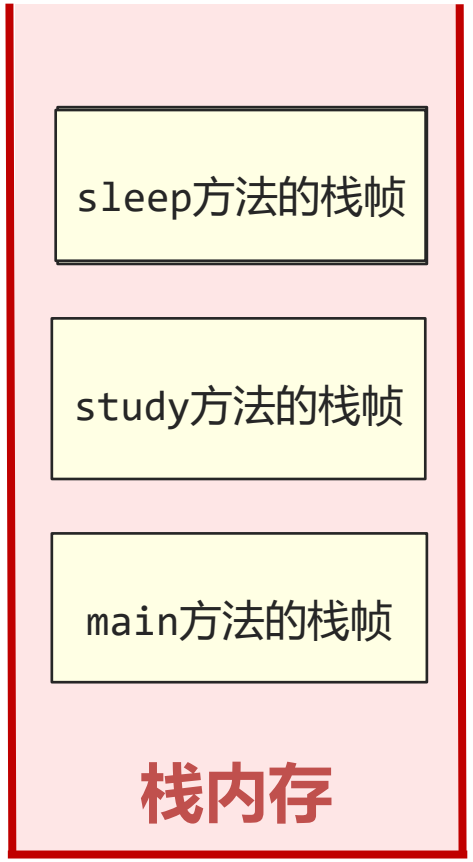
- Java虚拟机在运行Java程序过程中管理的内存区域，称之为运行时数据区。
- 《Java虚拟机规范》中规定了每一部分的作用。



Java虚拟机栈

Java虚拟机栈（Java Virtual Machine Stack）采用栈的数据结构来管理方法调用中的基本数据，先进后出（First In Last Out）,每一个方法的调用使用一个栈帧（Stack Frame）来保存。

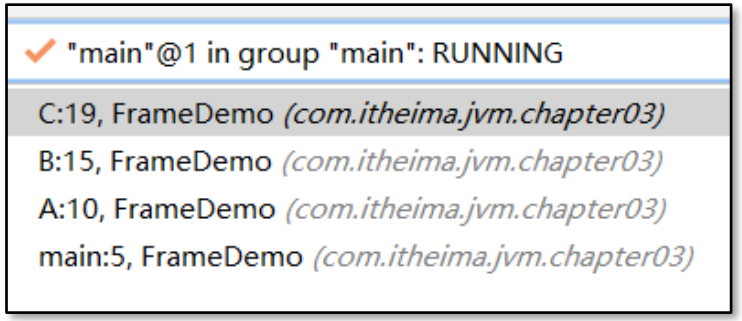
```
public class MethodDemo {
    public static void main(String[] args) {
        study();
    }

    public static void study(){

        eat();

        sleep();

    }

    public static void eat(){
        System.out.println("吃饭");
    }

    public static void sleep(){
        System.out.println("睡觉");
    }
}
```

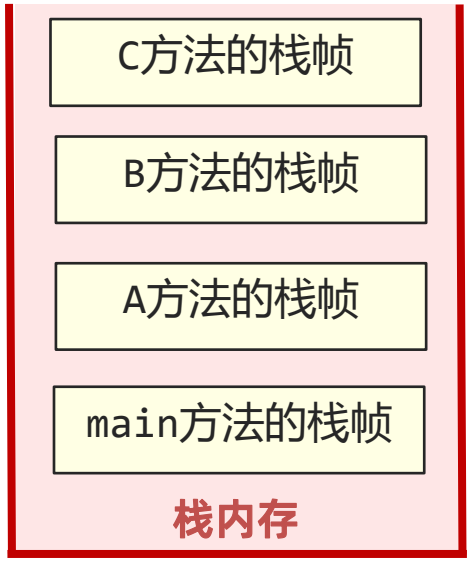


案例 通过Idea的debug工具查看栈帧的内容

```
public class FrameDemo {
    public static void main(String[] args) {
        A();
    }

    public static void A() {
        System.out.println("A执行了...");
        B();
    }

    public static void B() {
        System.out.println("B执行了...");
        C();
    }

    public static void C() {
        System.out.println("C执行了...");
    }
}
```

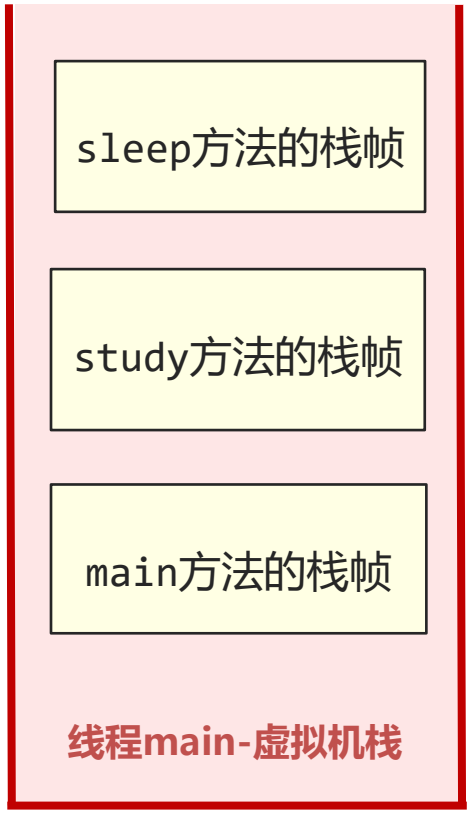
栈帧

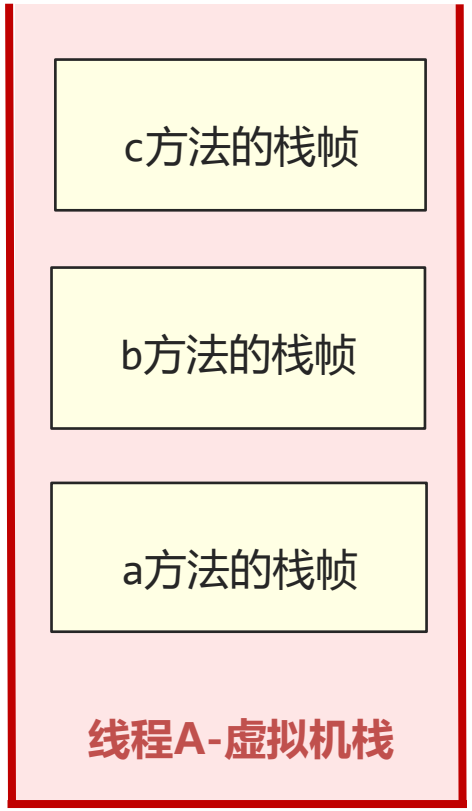




Java虚拟机栈

- Java虚拟机栈随着线程的创建而创建，而回收则会在线程的销毁时进行。由于方法可能会在不同线程中执行，每个线程都会包含一个自己的虚拟机栈。





Java虚拟机栈 - 栈帧的组成

局部变量表

局部变量表的作用是在运行过程中存放所有的局部变量

操作数栈

操作数栈是栈帧中虚拟机在执行指令过程中用来存放临时数据的一块区域

帧数据

帧数据主要包含动态链接、方法出口、异常表的引用

Java虚拟机栈-局部变量表

- 局部变量表的作用是在方法执行过程中存放所有的局部变量。编译成字节码文件时就可以确定局部变量表的内容。

```
public class Demo1 {
    public static void test1(){
        int i = 0;
        long j = 1;
    }
}
```

源代码

编译

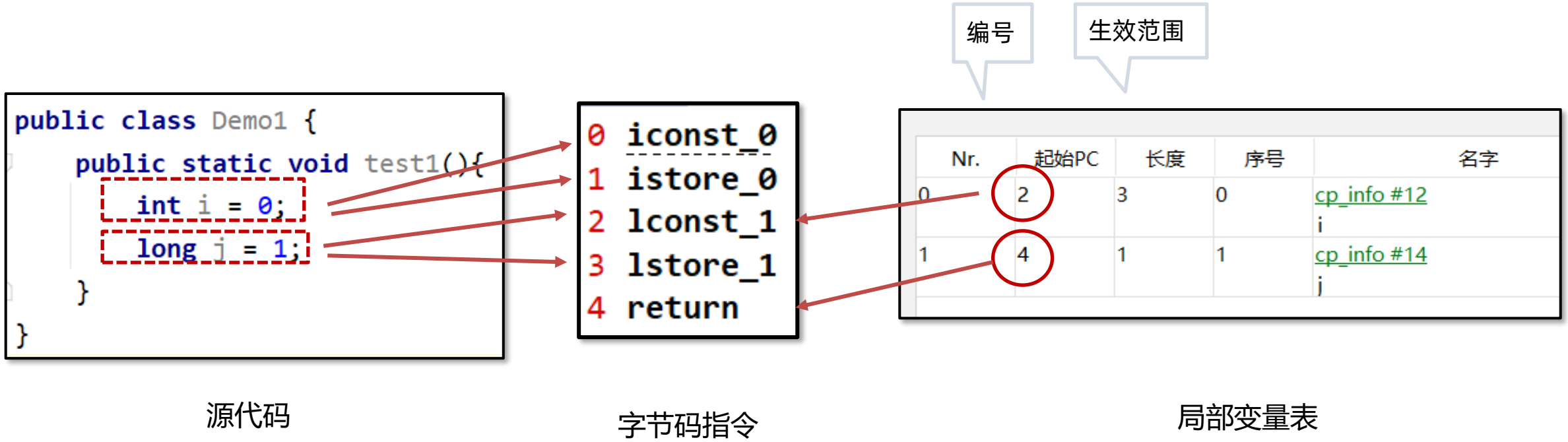
编号　　生效范围

Nr.	起始PC	长度	序号	名字
0	2	3	0	cp_info #12 i
1	4	1	1	cp_info #14 j

局部变量表

Java虚拟机栈-局部变量表

- 局部变量表的作用是在方法执行过程中存放所有的局部变量。编译成字节码文件时就可以确定局部变量表的内容。



Java虚拟机栈-局部变量表

- 栈帧中的局部变量表是一个数组，数组中每一个位置称之为槽(slot)，long和double类型占用两个槽，其他类型占用一个槽。

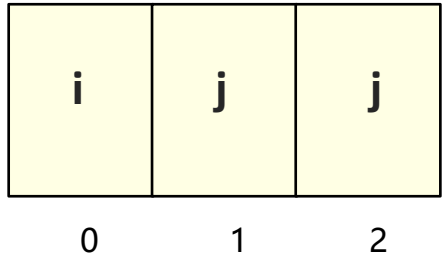
```
public class Demo1 {
    public static void test1(){
        int i = 0;
        long j = 1;
    }
}
```

源代码

槽的起始编号

Nr.	起始PC	长度	序号	名字
0	2	3	0	cp_info #12 i
1	4	1	1	cp_info #14 j

字节码文件中的局部变量表



栈帧中的局部变量表

Java虚拟机栈-局部变量表

- 实例方法中的序号为0的位置存放的是this，指的是当前调用方法的对象，运行时会在内存中存放实例对象的地址。

```
public void test2(){
    int i = 0;
    long j = 1;
}
```

源代码

编译

Nr.	起始PC	长度	序号	名字
0	0	5	0	cp_info #9 this
1	2	3	1	cp_info #12 i
2	4	1	2	cp_info #14 j

字节码文件中的局部变量表

Java虚拟机栈-局部变量表

- 方法参数也会保存在局部变量表中，其顺序与方法中参数定义的顺序一致。
- 局部变量表保存的内容有：实例方法的this对象，方法的参数，方法体中声明的局部变量。

```
public void test3(int k,int m){
    int i = 0;
    long j = 1;
}
```

源代码

编译

Nr.	起始PC	长度	序号	名字
0	0	6	0	cp_info #9 this
1	0	6	1	cp_info #19 k
2	0	6	2	cp_info #20 m
3	2	4	3	cp_info #12 i
4	5	1	4	cp_info #14 j

字节码文件中的局部变量表

思考

问题 以下代码的局部变量表中会占用几个槽?

```
public void test4(int k,int m){
    {
        int a = 1;
        int b = 2;
    }
    {
        int c = 1;
    }
    int i = 0;
    long j = 1;
}
```

字节码 | 异常表 | 杂项

操作数栈最大深度: 2

局部变量最大槽数: 6

字节码长度: 13

Java虚拟机栈-字节码中的局部变量

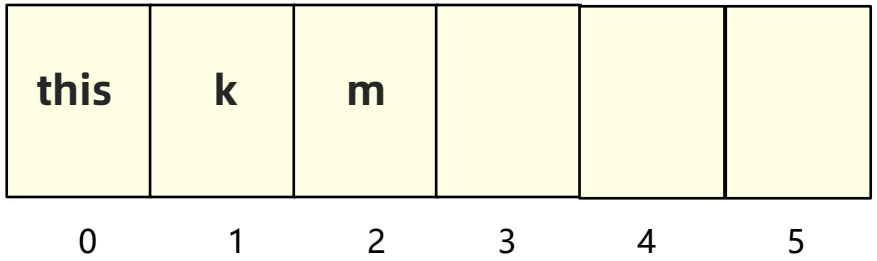
- 为了节省空间，局部变量表中的槽是可以复用的，一旦某个局部变量不再生效，当前槽就可以再次被使用。

```
public void test4(int k,int m){
    {
        int a = 1;
        int b = 2;
    }
    {
        int c = 1;
    }
    int i = 0;
    long j = 1;
}
```

源代码

```
 0 iconst_1
 1 istore_3
 2 iconst_2
 3 istore 4
 5 iconst_1
 6 istore_3
 7 iconst_0
 8 istore_3
 9 lconst_1
10 lstore 4
12 return
```

字节码指令



Java虚拟机栈-字节码中的局部变量

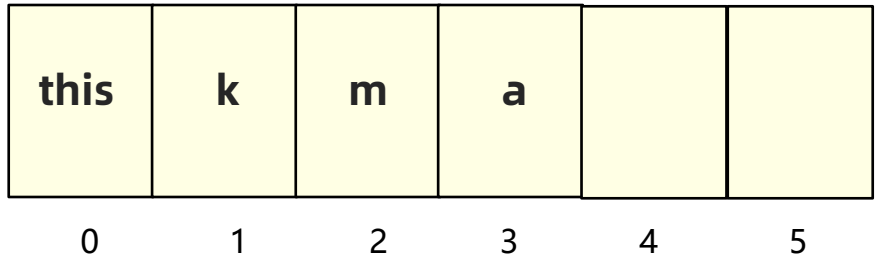
- 为了节省空间，局部变量表中的槽是可以复用的，一旦某个局部变量不再生效，当前槽就可以再次被使用。

```
public void test4(int k,int m){
    {
        int a = 1;
        int b = 2;
    }
    {
        int c = 1;
    }
    int i = 0;
    long j = 1;
}
```

源代码

```
 0 iconst_1
 1 istore_3
 2 iconst_2
 3 istore 4
 5 iconst_1
 6 istore_3
 7 iconst_0
 8 istore_3
 9 lconst_1
10 lstore 4
12 return
```

字节码指令



Java虚拟机栈-字节码中的局部变量

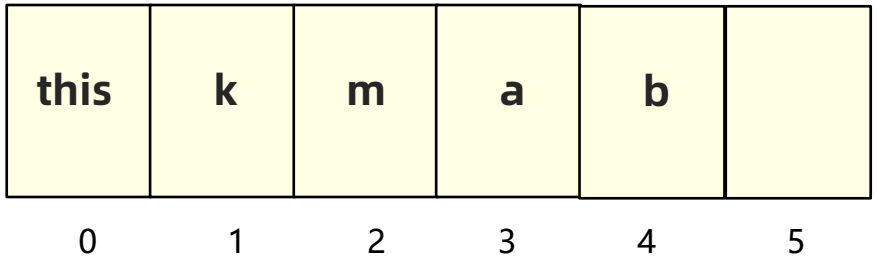
- 为了节省空间，局部变量表中的槽是可以复用的，一旦某个局部变量不再生效，当前槽就可以再次被使用。

```
public void test4(int k,int m){
    {
        int a = 1;
        int b = 2;
    }
    {
        int c = 1;
    }
    int i = 0;
    long j = 1;
}
```

源代码

```
 0 iconst_1
 1 istore_3
 2 iconst_2
 3 istore 4
 5 iconst_1
 6 istore_3
 7 iconst_0
 8 istore_3
 9 lconst_1
10 lstore 4
12 return
```

字节码指令



Java虚拟机栈-字节码中的局部变量

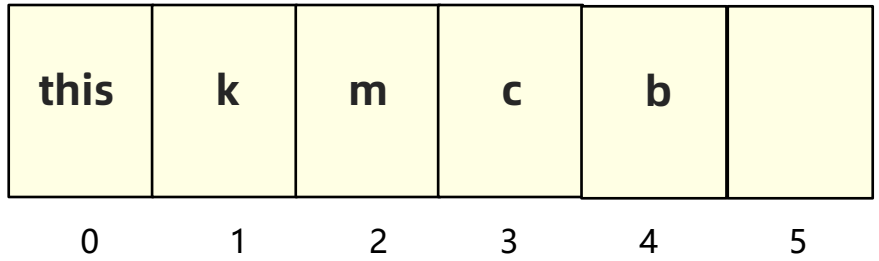
- 为了节省空间，局部变量表中的槽是可以复用的，一旦某个局部变量不再生效，当前槽就可以再次被使用。

```
public void test4(int k,int m){
    {
        int a = 1;
        int b = 2;
    }
    {
        int c = 1;
    }
    int i = 0;
    long j = 1;
}
```

源代码

```
 0 iconst_1
 1 istore_3
 2 iconst_2
 3 istore 4
 5 iconst_1
 6 istore_3
 7 iconst_0
 8 istore_3
 9 lconst_1
10 lstore 4
12 return
```

字节码指令



Java虚拟机栈-字节码中的局部变量

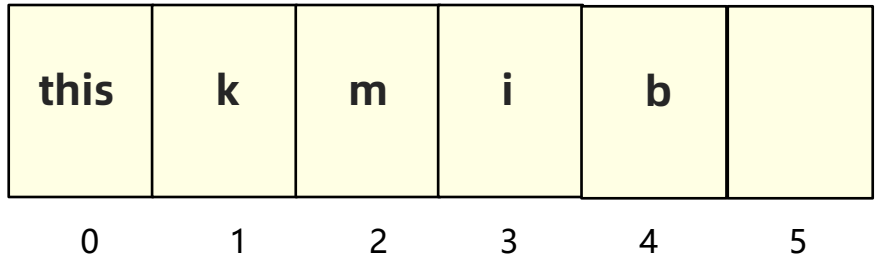
- 为了节省空间，局部变量表中的槽是可以复用的，一旦某个局部变量不再生效，当前槽就可以再次被使用。

```
public void test4(int k,int m){
    {
        int a = 1;
        int b = 2;
    }
    {
        int c = 1;
    }
    int i = 0;
    long j = 1;
}
```

源代码

```
 0 iconst_1
 1 istore_3
 2 iconst_2
 3 istore 4
 5 iconst_1
 6 istore_3
 7 iconst_0
 8 istore_3
 9 lconst_1
10 lstore 4
12 return
```

字节码指令



Java虚拟机栈-字节码中的局部变量

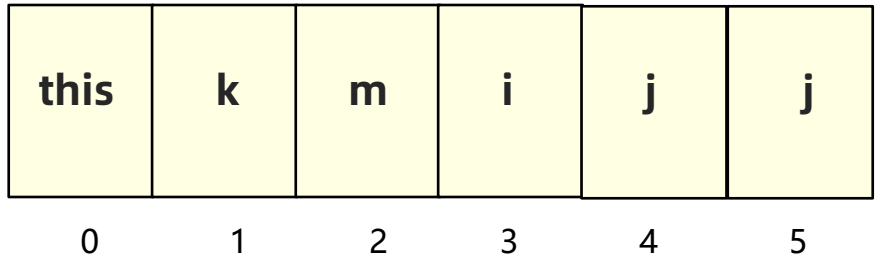
- 为了节省空间，局部变量表中的槽是可以复用的，一旦某个局部变量不再生效，当前槽就可以再次被使用。

```
public void test4(int k,int m){
    {
        int a = 1;
        int b = 2;
    }
    {
        int c = 1;
    }
    int i = 0;
    long j = 1;
}
```

源代码

```
 0 iconst_1
 1 istore_3
 2 iconst_2
 3 istore 4
 5 iconst_1
 6 istore_3
 7 iconst_0
 8 istore_3
 9 lconst_1
10 lstore 4
12 return
```

字节码指令



Java虚拟机栈 - 栈帧的组成

局部变量表

局部变量表的作用是在运行过程中存放所有的局部变量

操作数栈

操作数栈是栈帧中虚拟机在执行指令过程中用来存放临时数据的一块区域

帧数据

帧数据主要包含动态链接、方法出口、异常表的引用

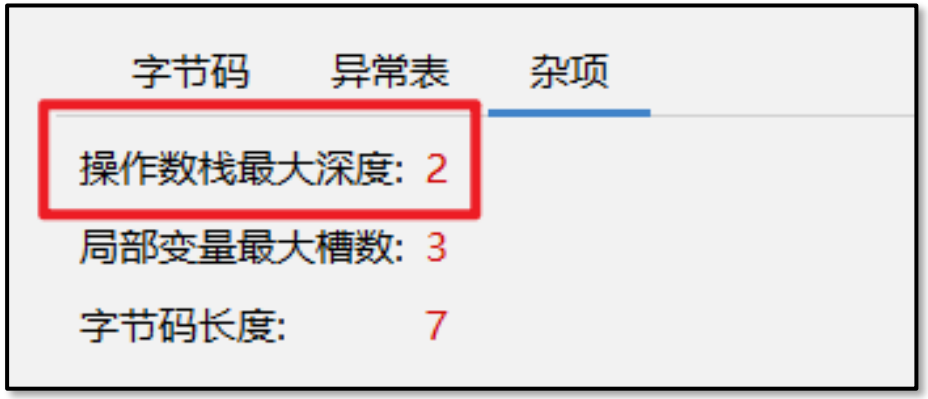
Java虚拟机栈-操作数栈

- 操作数栈是栈帧中虚拟机在执行指令过程中用来存放中间数据的一块区域。他是一种栈式的数据结构，如果一条指令将一个值压入操作数栈，则后面的指令可以弹出并使用该值。
- 在编译期就可以确定操作数栈的最大深度，从而在执行时正确的分配内存大小。

```
public void test5(){
    int i = 0;
    int j = i + 1;
}
```

源代码

编译



操作数栈的深度

案例 加法运算中操作数栈的应用

```
int i = 0;
int j = i + 1;
```

源代码

```
iconst_0
istore_1
iload_1
iconst_1
iadd
istore_2
return
```

字节码指令

将常量0放入操作数栈

从操作数栈取出放入局部变量表1号位置

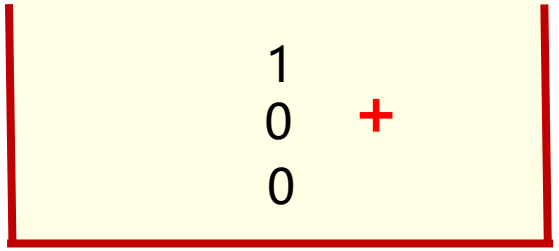
将局部变量表1中的数据放入操作数栈

将常量1放入操作数栈

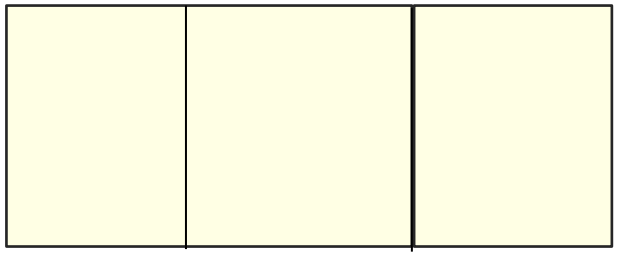
将操作数栈顶部的两个数据进行累加,结果放入栈中

局部变量表2号位置

方法结束，返回



操作数栈



数组下标 0 1(局部变量i) 2(局部变量j)

Java虚拟机栈 - 栈帧的组成

局部变量表

局部变量表的作用是在运行过程中存放所有的局部变量

操作数栈

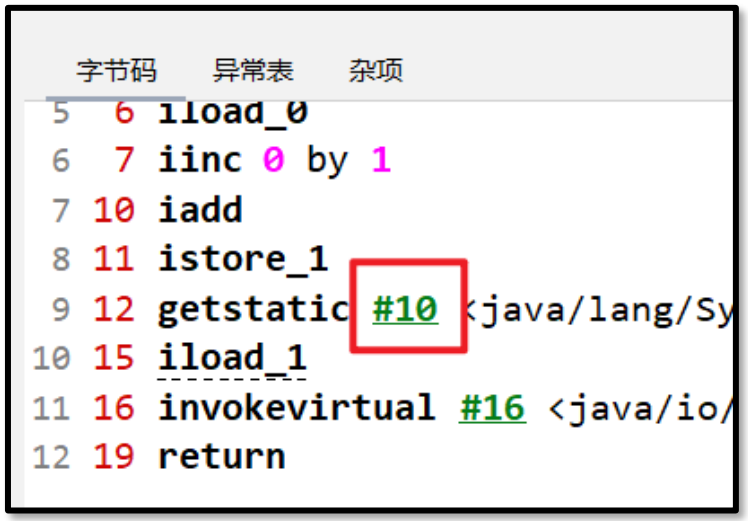
操作数栈是栈帧中虚拟机在执行指令过程中用来存放临时数据的一块区域

帧数据

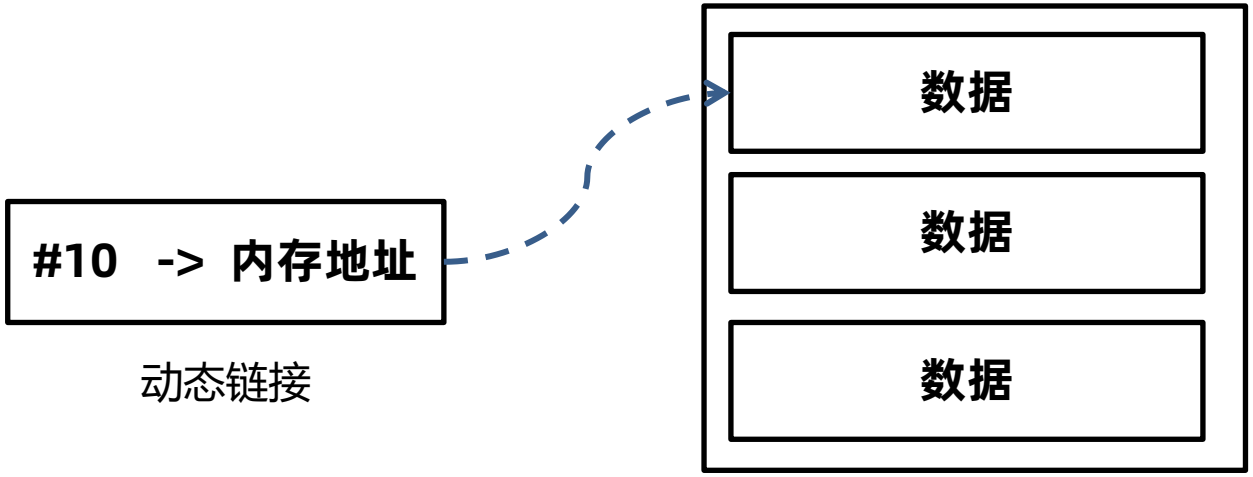
帧数据主要包含动态链接、方法出口、异常表的引用

Java虚拟机栈-帧数据

- **当前类的字节码指令引用了其他类的属性或者方法时，需要将符号引用（编号）转换成对应的运行时常量池中的内存地址。动态链接就保存了编号到运行时常量池的内存地址的映射关系。**



字节码指令

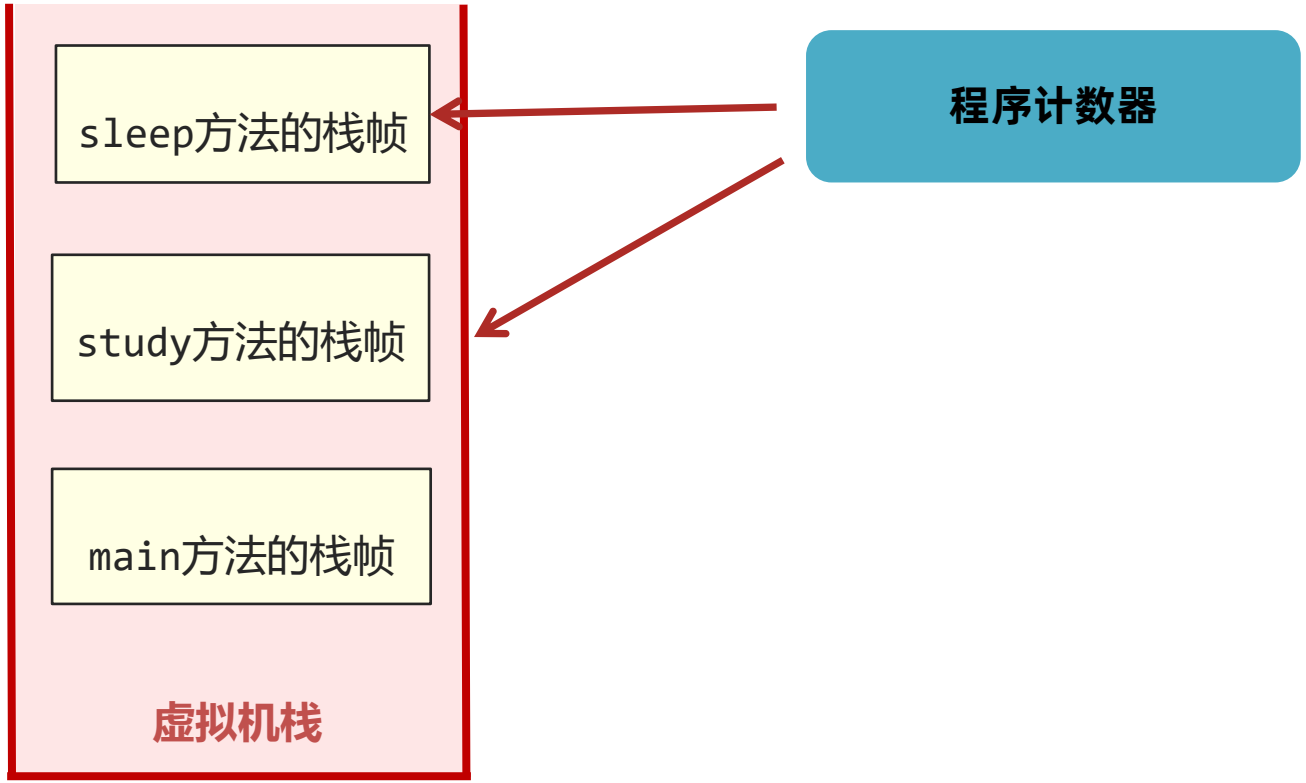


动态链接

运行时常量池

Java虚拟机栈-帧数据

- **方法出口指的是方法在正确或者异常结束时，当前栈帧会被弹出，同时程序计数器应该指向上一个栈帧中的下一条指令的地址。所以在当前栈帧中，需要存储此方法出口的地址。**



Java虚拟机栈-帧数据

- **异常表存放的是代码中异常的处理信息，包含了异常捕获的生效范围以及异常发生后跳转到的字节码指令位置。**

```
public static void test6()  {
    int i = 0;
    try{
        i = 1;
    }catch (Exception e){
        i = 2;
    }

}
```

源代码

编译

字节码 | 异常表 | 杂项

Nr.	起始PC	结束PC	跳转PC	捕获类型
0	2	4	7	cp_info #7 java/lang/Exception

异常表

Java虚拟机栈-帧数据

- **异常表存放的是代码中异常的处理信息，包含了异常捕获的生效范围以及异常发生后跳转到的字节码指令位置。**

```
public static void test6()  {
    int i = 0;
    try{
        i = 1;
    }catch (Exception e){
        i = 2;
    }

}
```

源代码

```
 0 iconst_0
 1 istore_0
 2 iconst_1
 3 istore_0
 4 goto 10 (+6)
 7 astore_1
 8 iconst_2
 9 istore_0
10 return
```

字节码指令

字节码 | 异常表 | 杂项

Nr.	起始PC	结束PC	跳转PC	捕获类型
0	2	4	7	cp_info #7 java/lang/Exception

异常表

Java虚拟机栈 - 栈帧的组成

局部变量表

局部变量表的作用是在运行过程中存放所有的局部变量

操作数栈

操作数栈是栈帧中虚拟机在执行指令过程中用来存放临时数据的一块区域

帧数据

帧数据主要包含动态链接、方法出口、异常表的引用

Java虚拟机栈-栈内存溢出

- Java虚拟机栈如果栈帧过多，占用内存超过栈内存可以分配的最大大小就会出现内存溢出。
- Java虚拟机栈内存溢出时会出现StackOverflowError的错误

Java虚拟机栈 – 默认大小

如果我们不指定栈的大小，JVM 将创建一个具有默认大小的栈。大小取决于操作系统和计算机的体系结构。

globals_linux_x86.hpp src/os_cpu/linux_x86/vm

```
29  // (see globals.hpp)
30
31  define_pd_global(bool, DontYieldALot,            false);
32  #ifdef AMD64
33  define_pd_global(intx, ThreadStackSize,          1024); // 0 => use system
34  define_pd_global(intx, VMThreadStackSize,        1024);
35  #else
36  // ThreadStackSize 320 allows a couple of test cases to run while
37  // keeping the number of threads that can be created high.  System
38  // default ThreadStackSize appears to be 512 which is too big.
39  define_pd_global(intx, ThreadStackSize,          320);
40  define_pd_global(intx, VMThreadStackSize,        512);
```

OpenJDK8虚拟机源码

01 Linux

x86（64 位）：1 MB

ppc：2 MB

02 BSD

x86（64 位）：1 MB

03 Solaris

64 位：1 MB

04 Windows

基于操作系统默认值

案例 Java虚拟机栈-栈内存溢出模拟

```
public static int count = 0;
//递归方法调用自己
public static void recursion(){
    System.out.println(++count);
    recursion();
}
```

需求：

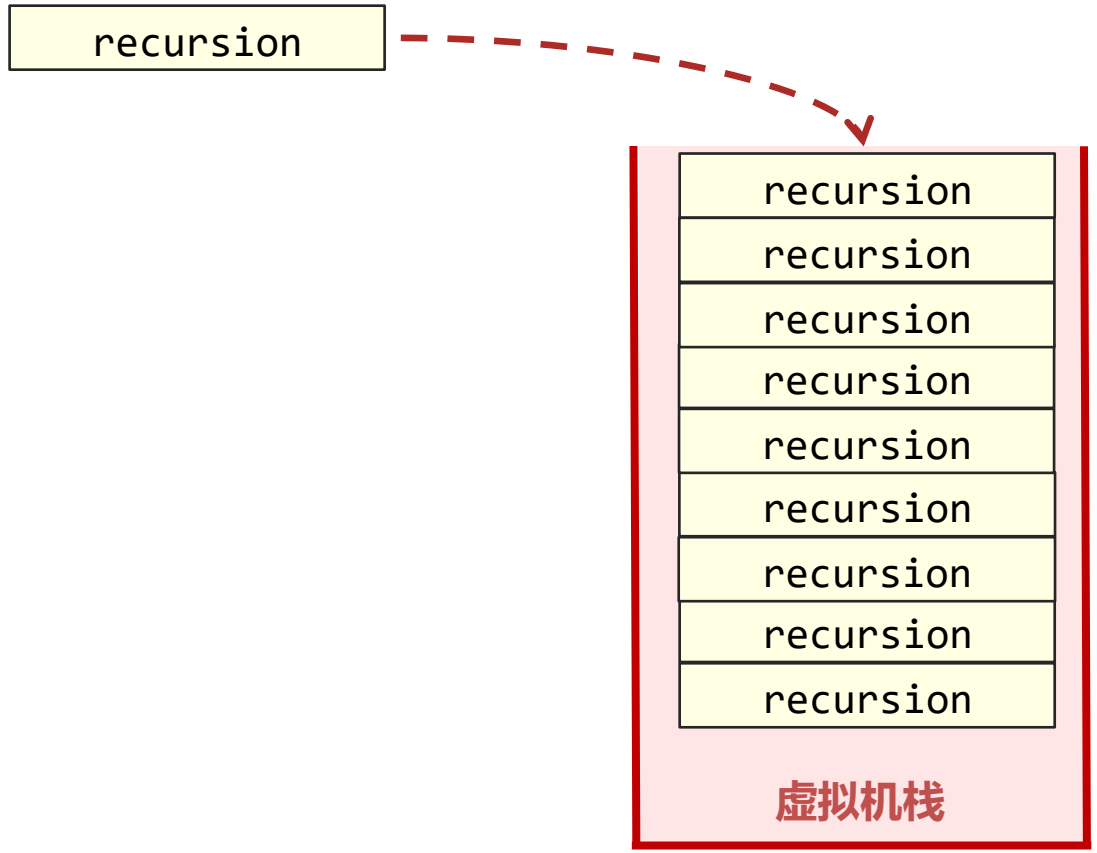
使用递归让方法调用自身，但是不设置退出条件。

定义调用次数的变量，每一次调用让变量加1。

查看错误发生时总调用的次数。

案例 Java虚拟机栈-栈内存溢出模拟

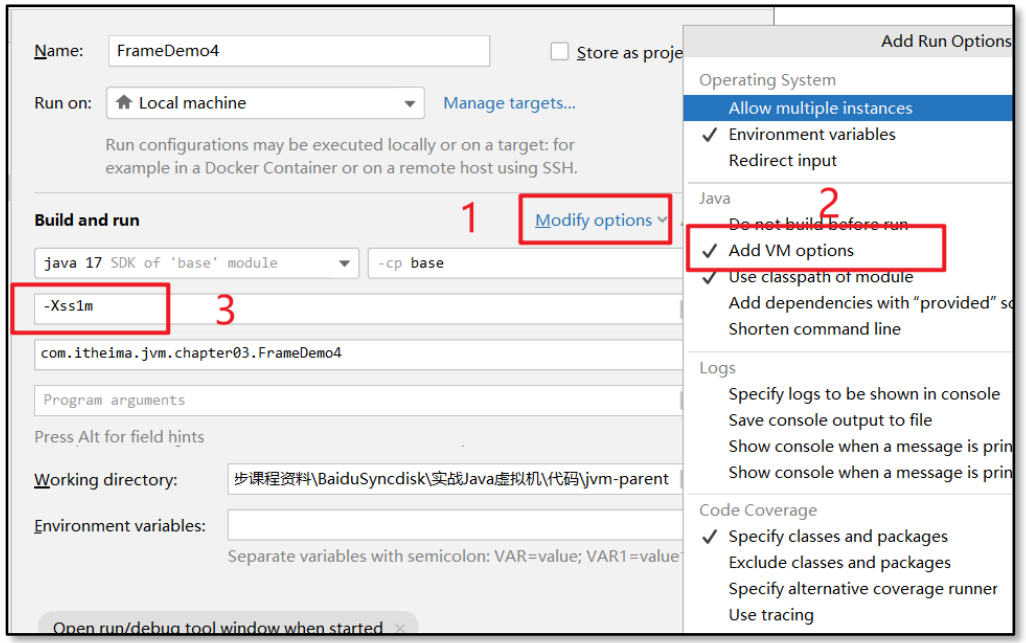
```
public static int count = 0;
//递归方法调用自己
public static void recursion(){
    System.out.println(++count);
    recursion();
}
```



Java虚拟机栈 – 设置大小

- **要修改Java虚拟机栈的大小，可以使用虚拟机参数 -Xss 。**
- 语法：-Xss栈大小
- 单位：字节（默认，必须是 1024 的倍数）、k或者K(KB)、m或者M(MB)、g或者G(GB)

```
-Xss1048576

-Xss1024K

-Xss1m

-Xss1g
```



Java虚拟机栈 – 注意事项

1、与-Xss类似，也可以使用 -XX:ThreadStackSize 调整标志来配置堆栈大小。

格式为： -XX:ThreadStackSize=1024

2、HotSpot JVM对栈大小的最大值和最小值有要求：

比如测试如下两个参数:

-Xss1k

-Xss1025m

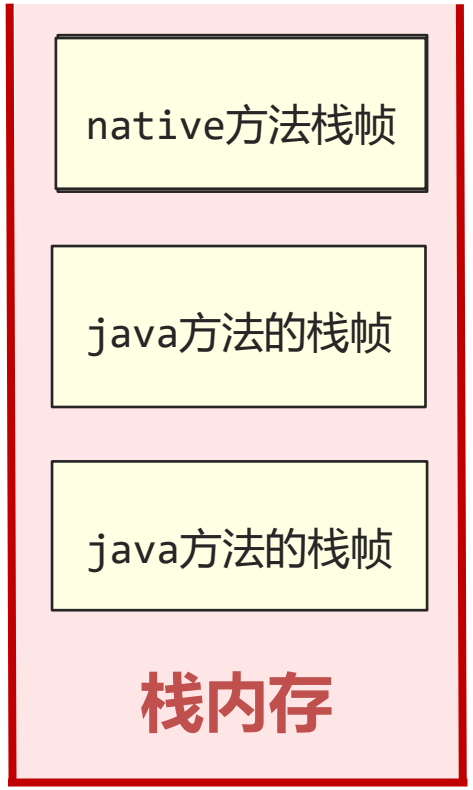
Windows（64位）下的JDK8测试最小值为180k，最大值为1024m。

3、局部变量过多、操作数栈深度过大也会影响栈内存的大小。

一般情况下，工作中即便使用了递归进行操作，栈的深度最多也只能到几百,不会出现栈的溢出。所以此参数可以手动指定为-Xss256k节省内存。

本地方法栈

- Java虚拟机栈存储了Java方法调用时的栈帧，而本地方法栈存储的是native本地方法的栈帧。
- 在Hotspot虚拟机中，Java虚拟机栈和本地方法栈实现上使用了同一个栈空间。本地方法栈会在栈内存上生成一个栈帧，临时保存方法的参数同时方便出现异常时也把本地方法的栈信息打印出来。



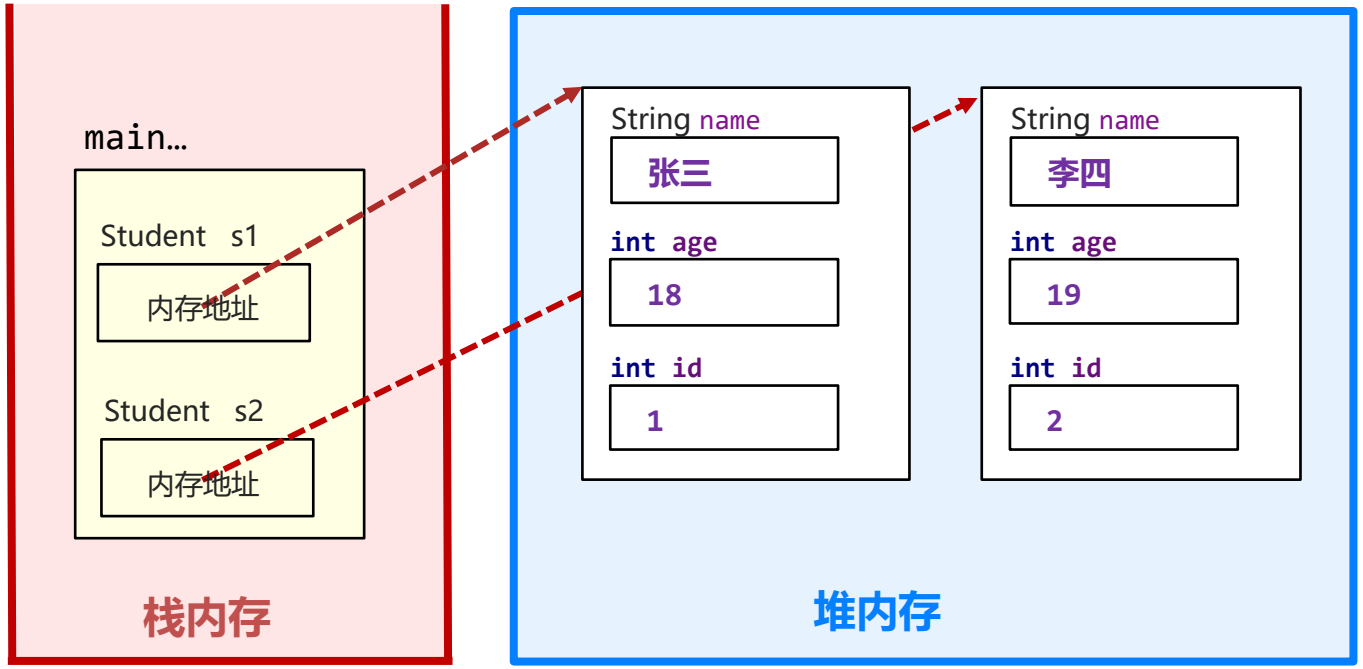
目录

Contents

Java堆

- 一般Java程序中堆内存是空间最大的一块内存区域。创建出来的对象都存在于堆上。
- 栈上的局部变量表中，可以存放堆上对象的引用。静态变量也可以存放堆对象的引用，通过静态变量就可以实现对象在线程之间共享。

```
public class Test {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "张三";
        s1.age = 18;
        s1.id = 1;
        s1.printTotalScore();
        s1.printAverageScore();

        Student s2 = new Student();
        s2.name = "李四";
        s2.age = 19;
        s2.id= 2;
        s2.printTotalScore();
        s2.printAverageScore();
    }
}
```



案例 实验-模拟堆区的溢出

需求：

通过new关键字不停创建对象，放入集合中，模拟堆内存的溢出，观察堆溢出之后的异常信息。

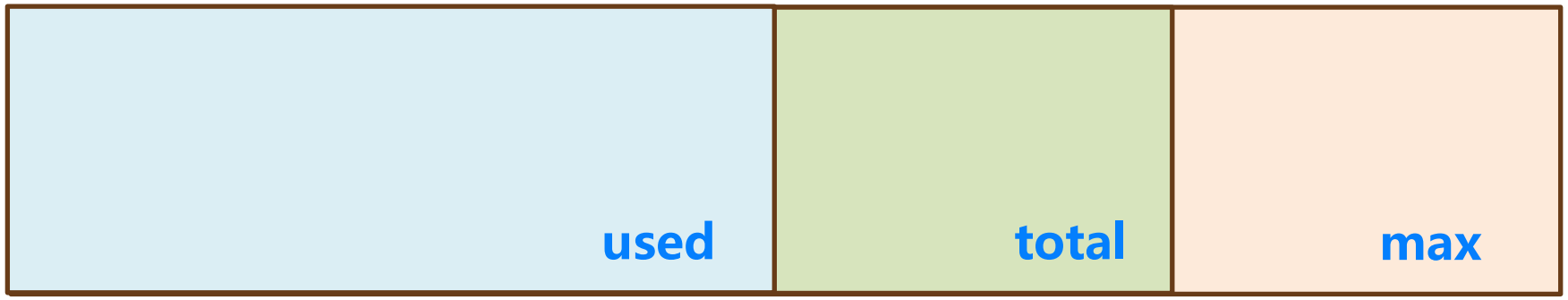
现象：

堆内存大小是有上限的，当对象一直向堆中放入对象达到上限之后，就会抛出OutOfMemory错误。

```
java.lang.OutOfMemoryError Create breakpoint : Java heap space
```

Java堆

- 堆空间有三个需要关注的值，used total max。
- used指的是当前已使用的堆内存，total是java虚拟机已经分配的可用堆内存，max是java虚拟机可以分配的最大堆内存。



arthas中堆内存相关的功能

- 堆内存used total max三个值可以通过dashboard命令看到。
- 手动指定刷新频率（不指定默认5秒一次）：dashboard –i 刷新频率(毫秒)

Memory	used	total	max	usage
heap	153M	981M	981M	15.68%
ps_eden_space	153M	256M	256M	60.11%
ps_survivor_space	0K	43520K	43520K	0.00%
ps_old_gen	0K	699392K	699392K	0.00%
nonheap	26M	27M	-1	97.10%
code_cache	5M	5M	240M	2.22%
metaspace	19M	19M	-1	96.86%
compressed_class_space	2M	2M	1024M	0.23%
direct	0K	0K	-	105.88%
mapped	0K	0K	-	0.00%
Runtime				

Java堆

- 随着堆中的对象增多，当total可以使用的内存即将不足时，java虚拟机会继续分配内存给堆。

Java堆

- 随着堆中的对象增多，当total可以使用的内存即将不足时，java虚拟机会继续分配内存给堆。

Java堆

- 如果堆内存不足，java虚拟机就会不断的分配内存，total值会变大。total最多只能与max相等。

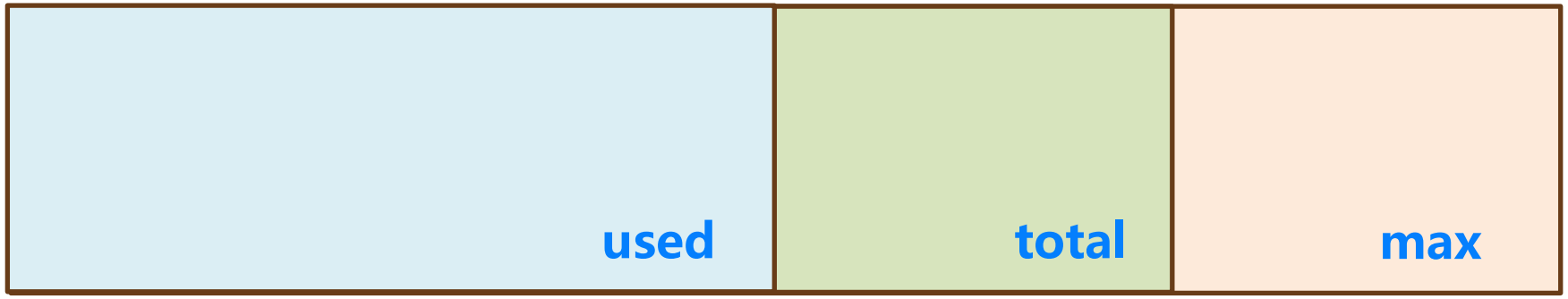
思考

问题	是不是当used = max = total的时候，堆内存就溢出了呢?

- 不是，堆内存溢出的判断条件比较复杂，在下一章《垃圾回收器》中会详细介绍。

Java堆

- 如果不设置任何的虚拟机参数，max默认是系统内存的1/4，total默认是系统内存的1/64。在实际应用中一般都需要设置total和max的值。
- Oracle官方文档：https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html



堆 – 设置大小

- **要修改堆的大小，可以使用虚拟机参数 -Xmx（max最大值）和-Xms (初始的total)。**
- 语法：-Xmx值 -Xms值
- 单位：字节（默认，必须是 1024 的倍数）、k或者K(KB)、m或者M(MB)、g或者G(GB)
- 限制：Xmx必须大于 2 MB，Xms必须大于1MB

```
-Xms6291456

-Xms6144k

-Xms6m

-Xmx83886080

-Xmx81920k

-Xmx80m
```

思考

问题 为什么arthas中显示的heap堆大小与设置的值不一样呢?

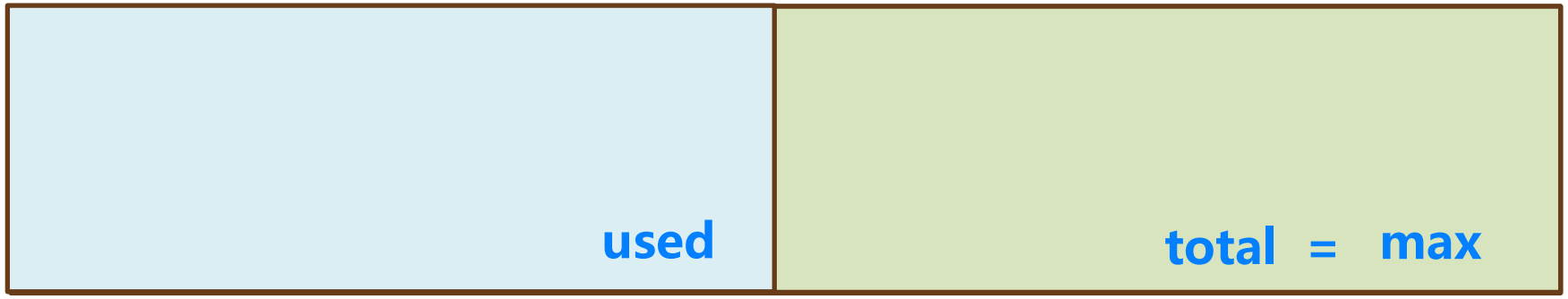
- arthas中的heap堆内存使用了JMX技术中内存获取方式，这种方式与垃圾回收器有关，计算的是可以分配对象的内存，而不是整个内存。

Memory	used	total	max	usage
heap	153M	981M	981M	15.68%
ps_eden_space	153M	256M	256M	60.11%
ps_survivor_space	0K	43520K	43520K	0.00%
ps_old_gen	0K	699392K	699392K	0.00%
nonheap	26M	27M	-1	97.10%
code_cache	5M	5M	240M	2.22%
metaspace	19M	19M	-1	96.86%
compressed_class_space	2M	2M	1024M	0.23%
direct	0K	0K	-	105.88%
mapped	0K	0K	-	0.00%

-Xmx1g -Xms1g

Java堆

- Java服务端程序开发时，建议将-Xmx和-Xms设置为相同的值，这样在程序启动之后可使用的总内存就是最大内存，而无需向java虚拟机再次申请，减少了申请并分配内存时间上的开销，同时也不会出现内存过剩之后堆收缩的情况。
- -Xmx具体设置的值与实际的应用程序运行环境有关，在《实战篇》中会给出设置方案。

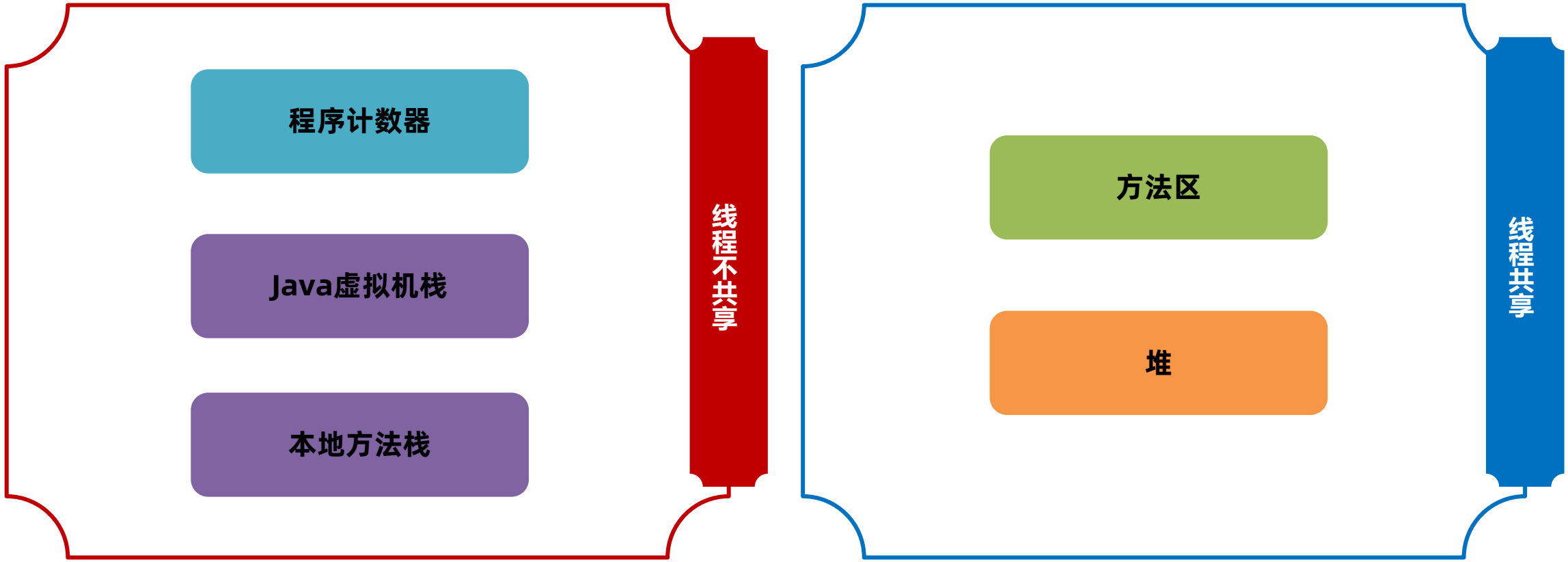


目录

Contents

运行时数据区-总览

- Java虚拟机在运行Java程序过程中管理的内存区域，称之为运行时数据区。
- 《Java虚拟机规范》中规定了每一部分的作用。



方法区（Method Area）

- 方法区是存放基础信息的位置，线程共享，主要包含三部分内容：

01

类的元信息

保存了所有类的基本信息

02

运行时常量池

保存了字节码文件中的常量池内容

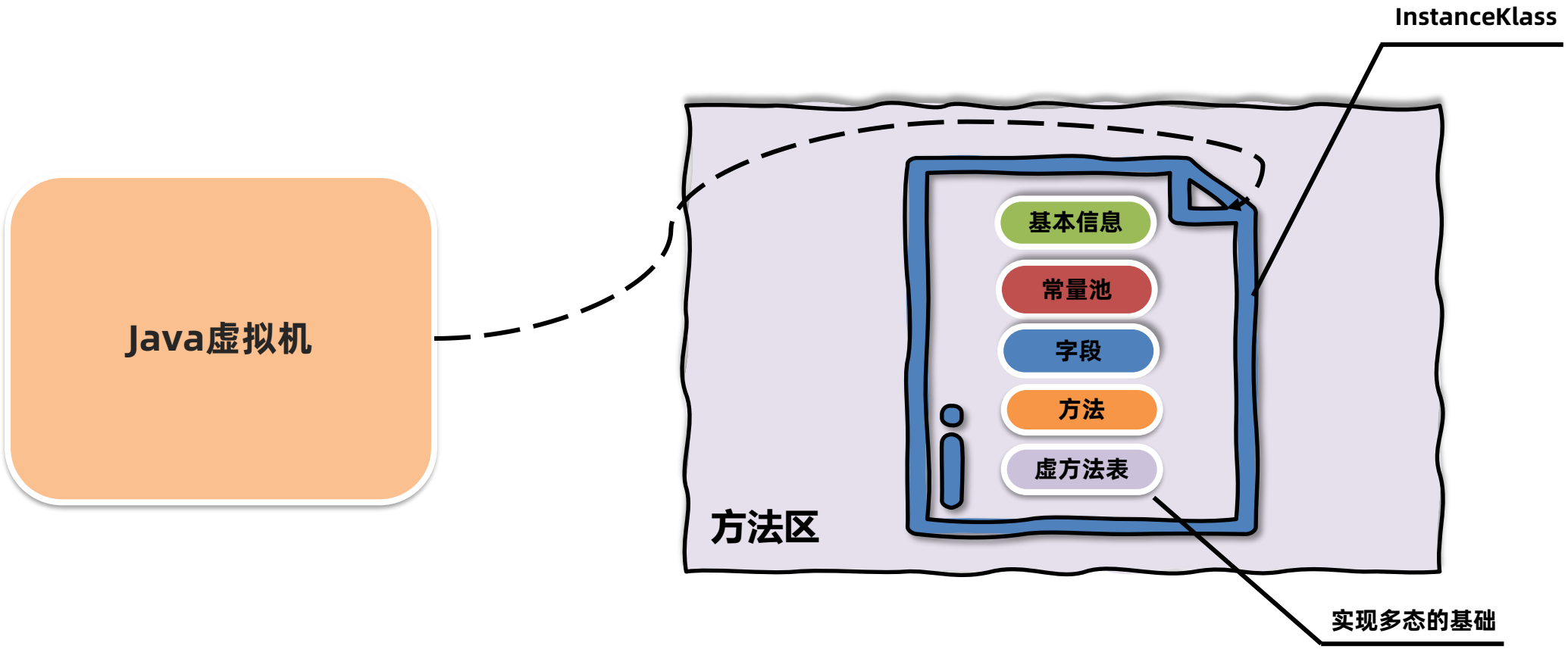
03

字符串常量池

保存了字符串常量

方法区（Method Area）

- 方法区是用来存储每个类的基本信息（元信息），一般称之为**InstanceKlass对象**。在类的加载阶段完成。



方法区（Method Area）

- 方法区是存放基础信息的位置，线程共享，主要包含三部分内容：

01

类的元信息

保存了所有类的基本信息

02

运行时常量池

保存了字节码文件中的常量池内容

03

字符串常量池

保存了字符串常量

方法区（Method Area）

- 方法区除了存储类的元信息之外，还存放了运行时常量池。常量池中存放的是字节码中的常量池内容。
- 字节码文件中通过编号查表的方式找到常量，这种常量池称为静态常量池。当常量池加载到内存中之后，可以通过内存地址快速的定位到常量池中的内容，这种常量池称为运行时常量池。

静态常量池

加载+连接

Constants

Index	Constant Type	Constant Value
1	JVM_CONSTANT_Methodref	#6 #30
2	JVM_CONSTANT_Class	public class HsdbDemo @0x00000007c0060828
3	JVM_CONSTANT_Methodref	#2 #30
4	JVM_CONSTANT_Fieldref	#32 #33
5	JVM_CONSTANT_Methodref	#34 #35
6	JVM_CONSTANT_Class	public class java.lang.Object @0x00000007c0000f28
7	JVM_CONSTANT_Utf8	"i"
8	JVM_CONSTANT_Utf8	"I"
9	JVM_CONSTANT_Utf8	"ConstantValue"

运行时常量池

方法区（Method Area）

- 方法区是《Java虚拟机规范》中设计的虚拟概念，每款Java虚拟机在实现上都各不相同。Hotspot设计如下：
  - JDK7及之前的版本将方法区存放在堆区域中的永久代空间，堆的大小由虚拟机参数来控制。
  - JDK8及之后的版本将方法区存放在元空间中，元空间位于操作系统维护的直接内存中，默认情况下只要不超过操作系统承受的上限，可以一直分配。

arthas中查看方法区

- 使用memory打印出内存情况，JDK7及之前的版本查看ps_perm_gen属性。
- JDK8及之后的版本查看metaspace属性。

```
[arthas@29656]$ memory
Memory                used                 total                max
heap                  107M                 487M                 7227M
ps_eden_space         107M                 127M                 2668M
ps_survivor_space     0K                   21504K               21504K
ps_old_gen            0K                   346624K              5550080K
nonheap               18M                  23M                  130M
code_cache            812K                 2496K                49152K
ps_perm_gen           17M                  21M                  82M
direct                0K                   0K                   -
mapped                0K                   0K                   -
```

JDK7- 永久代

```
[arthas@28764]$ memory
Memory                used                 total                max
heap                  125M                 489M                 7227M
ps_eden_space         125M                 128M                 2668M
ps_survivor_space     0K                   21504K               21504K
ps_old_gen            0K                   348160K              5550080K
nonheap               26M                  27M                  -1
code_cache            5M                   5M                   240M
metaspace             19M                  19M                  -1
compressed_class_space 2M                  2M                   1024M
direct                0K                   0K                   -
mapped                0K                   0K                   -
```

JDK8- 元空间

案例

实验-模拟方法区的溢出

需求：

通过ByteBuddy框架，动态生成字节码数据，加载到内存中。通过死循环不停地加载到方法区，观察方法区是否会出现内存溢出的情况。分别在JDK7和JDK8上运行上述代码。

ByteBuddy框架的基本使用方法

ByteBuddy是一个基于Java的开源库，用于生成和操作Java字节码。

1.引入依赖

```
<dependency>
   <groupId>net.bytebuddy</groupId>
   <artifactId>byte-buddy</artifactId>
   <version>1.12.23</version>
 </dependency>
```

2.创建ClassWriter对象

```
ClassWriter classWriter = new ClassWriter(0);
```

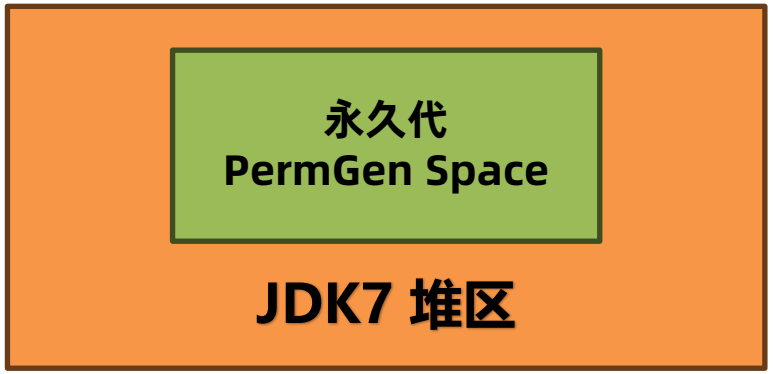
3.调用visit方法，创建字节码数据。

```
classWriter.visit(Opcodes.V1_7,Opcodes.ACC_PUBLIC,name,null ,"java/lang/Object",null);
byte[] bytes = classWriter.toByteArray();
```

方法区（Method Area）溢出

实验发现，JDK7上运行大概十几万次，就出现了错误。在JDK8上运行百万次，程序都没有出现任何错误，但是内存会直线升高。这说明JDK7和JDK8在方法区的存放上，采用了不同的设计。

- JDK7将方法区存放在堆区域中的永久代空间，堆的大小由虚拟机参数-XX:MaxPermSize=值来控制。
- JDK8将方法区存放在元空间中，元空间位于操作系统维护的直接内存中，默认情况下只要不超过操作系统承受的上限，可以一直分配。可以使用-XX:MaxMetaspaceSize=值将元空间最大大小进行限制。



方法区（Method Area）

- 方法区是存放基础信息的位置，线程共享，主要包含三部分内容：

01

类的元信息

保存了所有类的基本信息

02

运行时常量池

保存了字节码文件中的常量池内容

03

字符串常量池

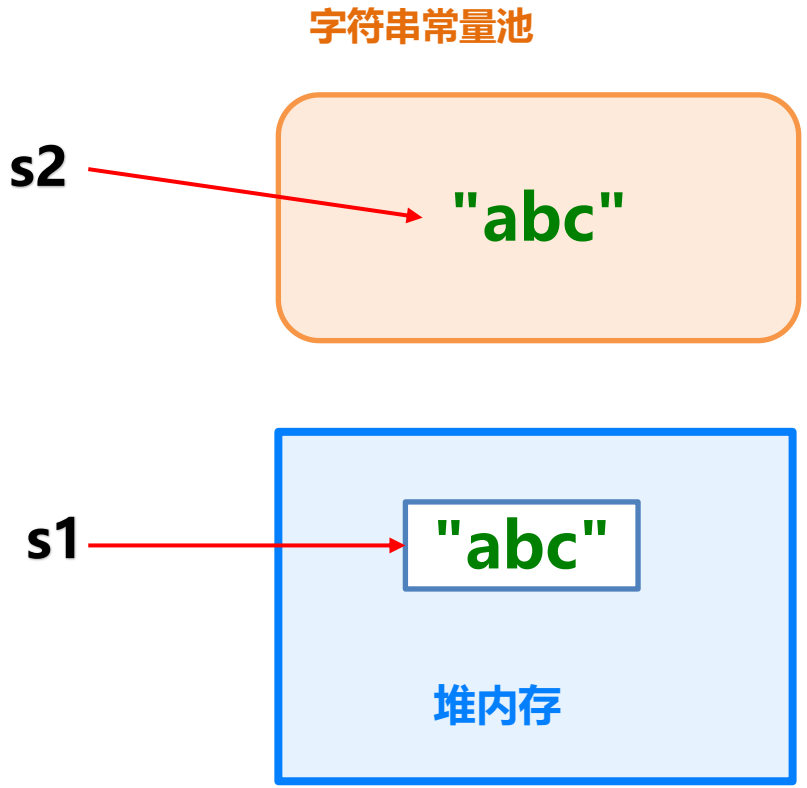
保存了字符串常量

方法区（Method Area）字符串常量池

- 方法区中除了类的元信息、运行时常量池之外，还有一块区域叫字符串常量池(**StringTable**)。
- 字符串常量池存储在代码中定义的常量字符串内容。比如“123” 这个123就会被放入字符串常量池。

```
public class Test {
    public static void main(String[] args) {
        String s1 = new String("abc");
        String s2 = "abc";
        System.out.println(s1 == s2);
    }
}
```

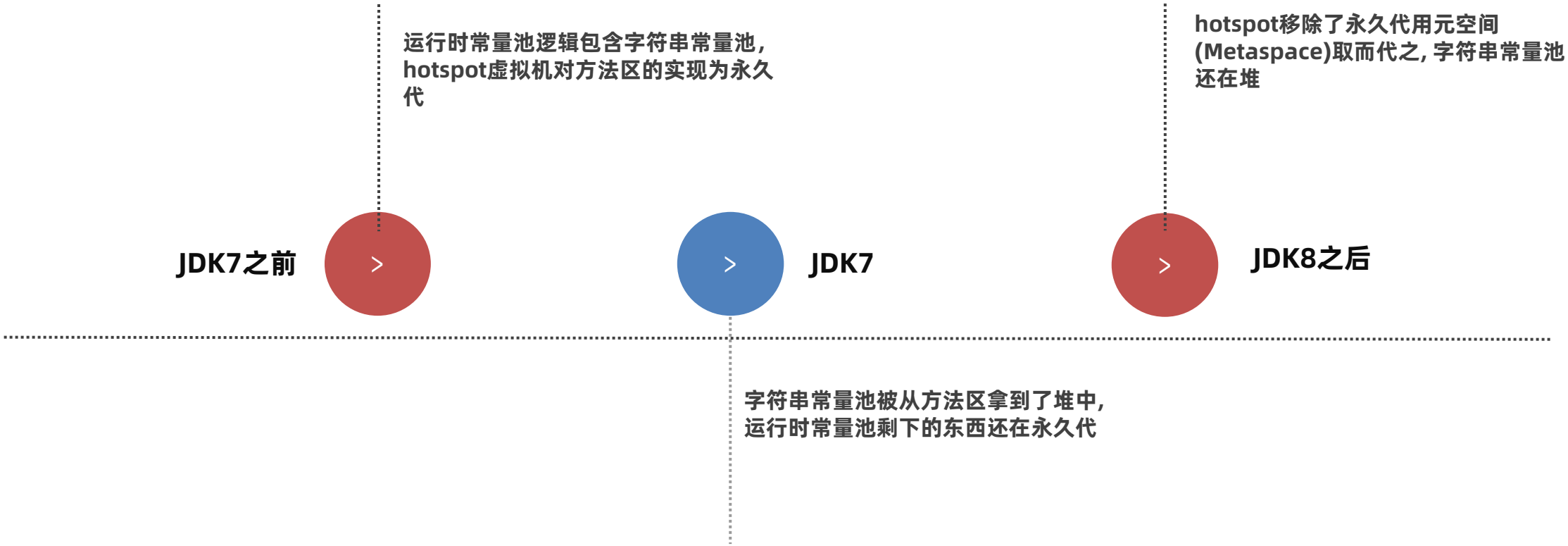
false



方法区（Method Area）字符串常量池(StringTable)

字符串常量池和运行时常量池有什么关系?

早期设计时，字符串常量池是属于运行时常量池的一部分，他们存储的位置也是一致的。后续做出了调整，将字符串常量池和运行时常量池做了拆分。



案例

StringTable的练习题1：

需求：

- 通过字节码指令分析如下代码的运行结果？

```
public static void main(String[] args) {
    String a = "1";
    String b = "2";
    String c = "12";
    String d = a + b;
    System.out.println(c == d);
}
```

案例 StringTable的练习题1：

```
public static void main(String[] args) {
    String a = "1";
    String b = "2";
    String c = "12";
    String d = a + b;
    System.out.println(c == d);
}
```

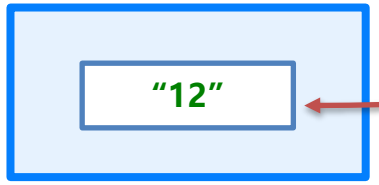
源码

```
 0 ldc #2 <1>
 2 astore_1
 3 ldc #3 <2>
 5 astore_2
 6 ldc #4 <12>
 8 astore_3
 9 new #5 <java/lang/StringBuilder>
12 dup
13 invokespecial #6 <java/lang/StringBuilder.<init> :
16 aload_1
17 invokevirtual #7 <java/lang/StringBuilder.append :
20 aload_2
21 invokevirtual #7 <java/lang/StringBuilder.append :
24 invokevirtual #8 <java/lang/StringBuilder.toString
27 astore 4
```

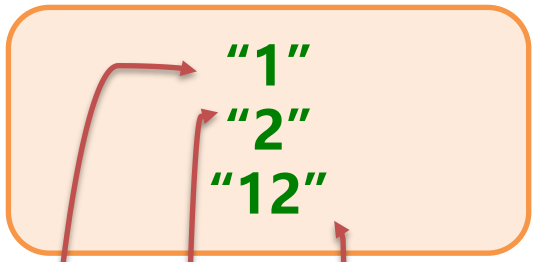
从常量池中获取字符串1的地址
放入操作数栈

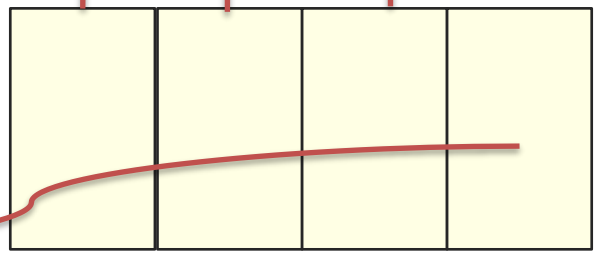
将操作数栈中的值放入局部变量表

字节码指令



堆内存





局部变量 a b c d

案例

StringTable的练习题2：

需求：

- 通过字节码指令分析如下代码的运行结果？

```
public static void main(String[] args) {
    String a = "1";
    String b = "2";
    String c = "12";
    String d = "1" + "2";
    System.out.println(c == d);
}
```

案例

StringTable的练习题：

需求：

- 通过字节码指令分析如下代码的运行结果？

```
public static void main(String[] args) {
    String a = "1";
    String b = "2";
    String c = "12";
    String d = a + b;
    System.out.println(c == d);
}
```

变量连接使用StringBuilder

```
public static void main(String[] args) {
    String a = "1";
    String b = "2";
    String c = "12";
    String d = "1" + "2";
    System.out.println(c == d);
}
```

常量，编译阶段直接连接

案例

神奇的intern

需求：

- String.intern()方法是可以手动将字符串放入字符串常量池中，分别在JDK6 JDK8下执行代码，JDK6 中结果是false false ， JDK8中是true false

```
public static void main(String[] args) {
    String s1 = new StringBuilder().append("think").append("123").toString();

    System.out.println(s1.intern() == s1);

    String s2 = new StringBuilder().append("ja").append("va").toString();

    System.out.println(s2.intern() == s2);
}
```

案例 神奇的intern

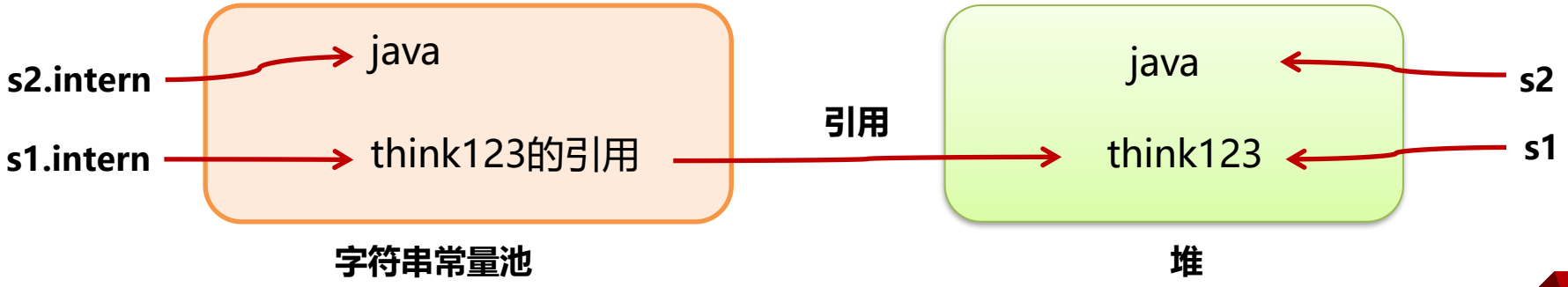
JDK6版本中intern () 方法会把第一次遇到的字符串实例复制到永久代的字符串常量池中，返回的也是永久代里面这个字符串实例的引用。JVM启动时就会把java加入到常量池中。

```
public static void main(String[] args) {
    String s1 = new StringBuilder().append("think").append("123").toString();

    System.out.println(s1.intern() == s1);

    String s2 = new StringBuilder().append("ja").append("va").toString();

    System.out.println(s2.intern() == s2);
}
```

案例 神奇的intern

JDK7及之后版本中由于字符串常量池在堆上，所以intern () 方法会把第一次遇到的字符串的引用放入字符串常量池。

```
public static void main(String[] args) {
    String s1 = new StringBuilder().append("think").append("123").toString();

    System.out.println(s1.intern() == s1);

    String s2 = new StringBuilder().append("ja").append("va").toString();

    System.out.println(s2.intern() == s2);
}
```

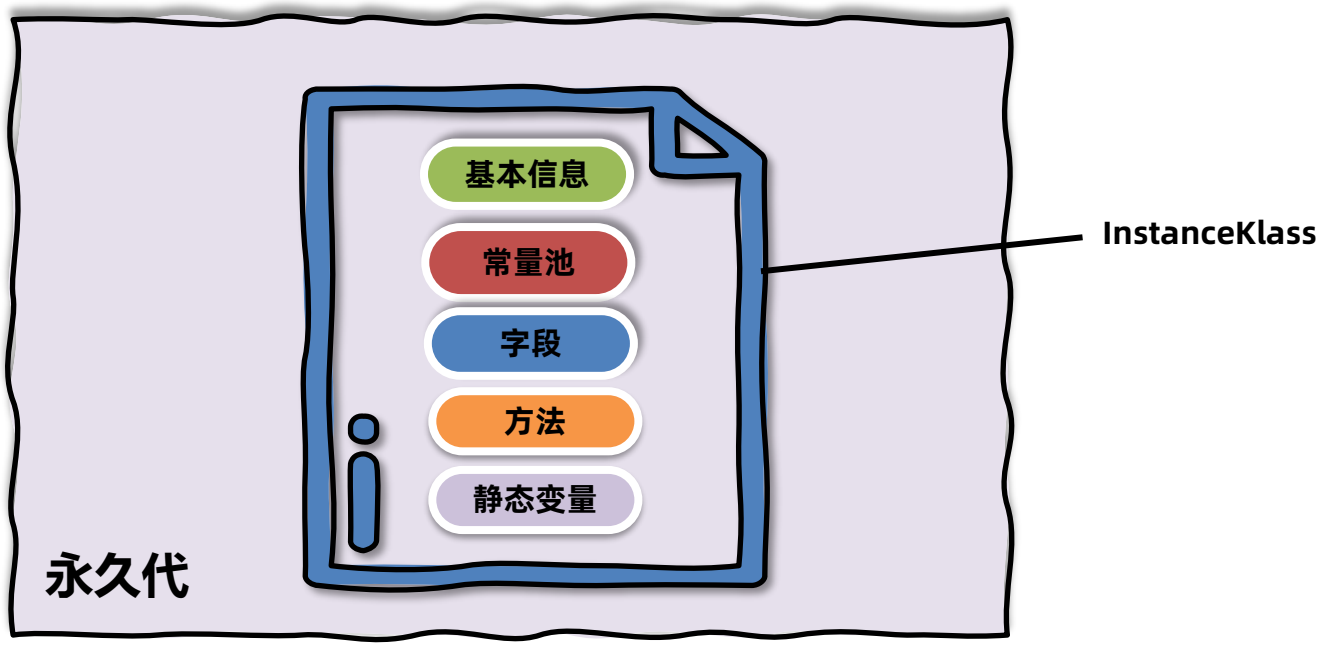


静态变量的存储

思考

问题	运行时数据区都学完了，静态变量存储在哪里呢？

- JDK6及之前的版本中，静态变量是存放在方法区中的，也就是永久代。

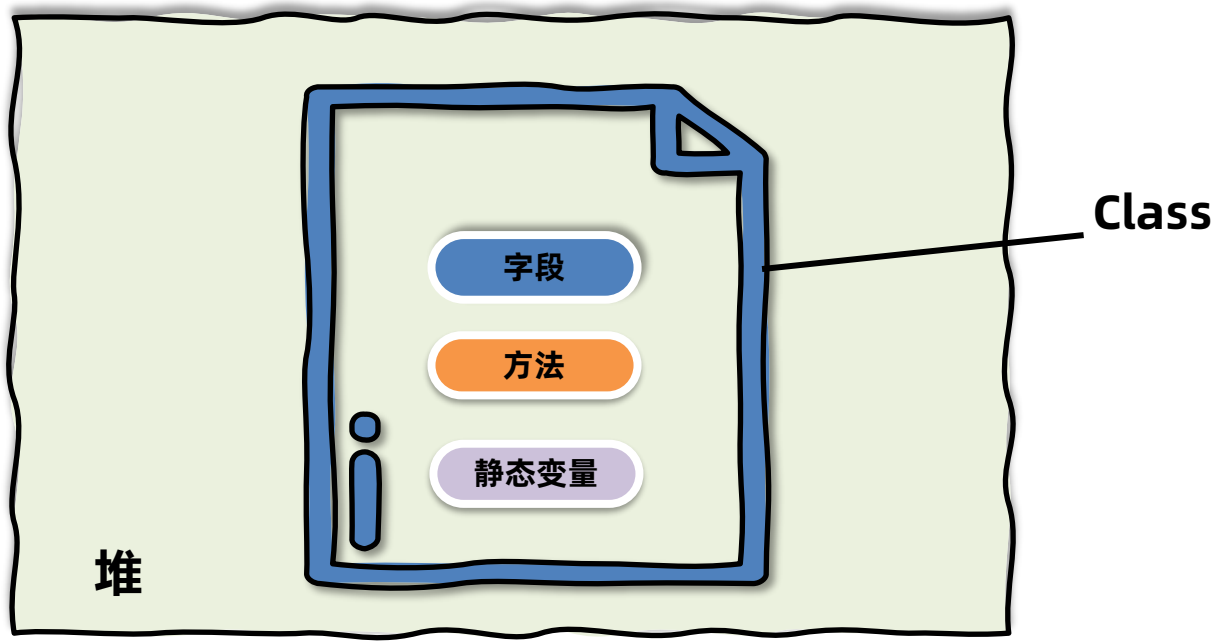


静态变量的存储

思考

- JDK7及之后的版本中，静态变量是存放在堆中的Class对象中，脱离了永久代。

具体源码可参考虚拟机源码：BytecodeInterpreter针对putstatic指令的处理。



目录

Contents

直接内存(Direct Memory)

- 直接内存（Direct Memory）并不在《Java虚拟机规范》中存在，所以并不属于Java运行时的内存区域。

在 JDK 1.4 中引入了 NIO 机制，使用了直接内存，主要为了解决以下两个问题:

1、Java堆中的对象如果不再使用要回收，回收时会影响对象的创建和使用。

2、IO操作比如读文件，需要先把文件读入直接内存（缓冲区）再把数据复制到Java堆中。

现在直接放入直接内存即可，同时Java堆上维护直接内存的引用，减少了数据复制的开销。写文件也是类似的思路。

直接内存(Direct Memory)

- **要创建直接内存上的数据，可以使用ByteBuffer。**
- 语法： `ByteBuffer directBuffer = ByteBuffer.allocateDirect(size);`
- 注意事项： arthas的memory命令可以查看直接内存大小，属性名direct。

Memory	used	total	max	usage
heap	43M	489M	7227M	0.60%
ps_eden_space	19M	128M	2668M	0.73%
ps_survivor_space	20M	21M	21M	99.90%
ps_old_gen	2M	340M	5420M	0.05%
nonheap	28M	29M	-1	97.37%
code_cache	6M	6M	240M	2.68%
metaspace	19M	20M	-1	97.28%
compressed_class_space	2M	2M	1024M	0.23%
direct	3300M	3300M	-	100.00%
mapped	0K	0K	-	0.00%

arthas memory命令(JDK8)

直接内存(Direct Memory)

- 如果需要手动调整直接内存的大小，可以使用-XX:MaxDirectMemorySize=大小

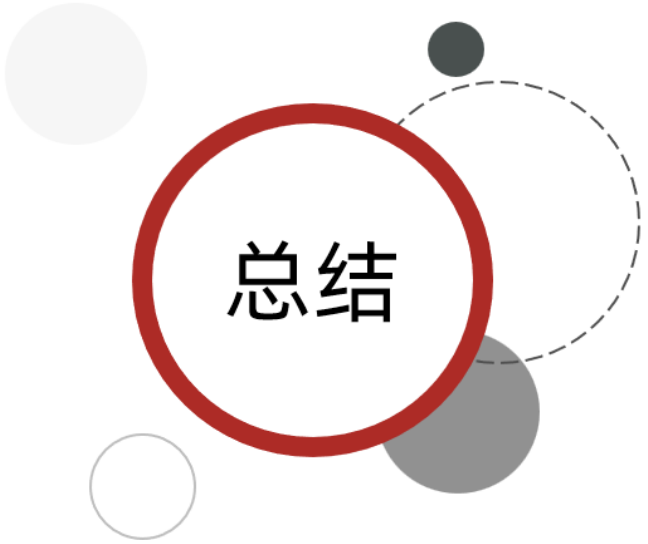
  单位k或K表示千字节，m或M表示兆字节，g或G表示千兆字节。默认不设置该参数情况下，JVM 自动选择 最大分配的大小。

以下示例以不同的单位说明如何将 直接内存大小设置为 1024 KB：

-XX:MaxDirectMemorySize=1m

-XX:MaxDirectMemorySize=1024k

-XX:MaxDirectMemorySize=1048576



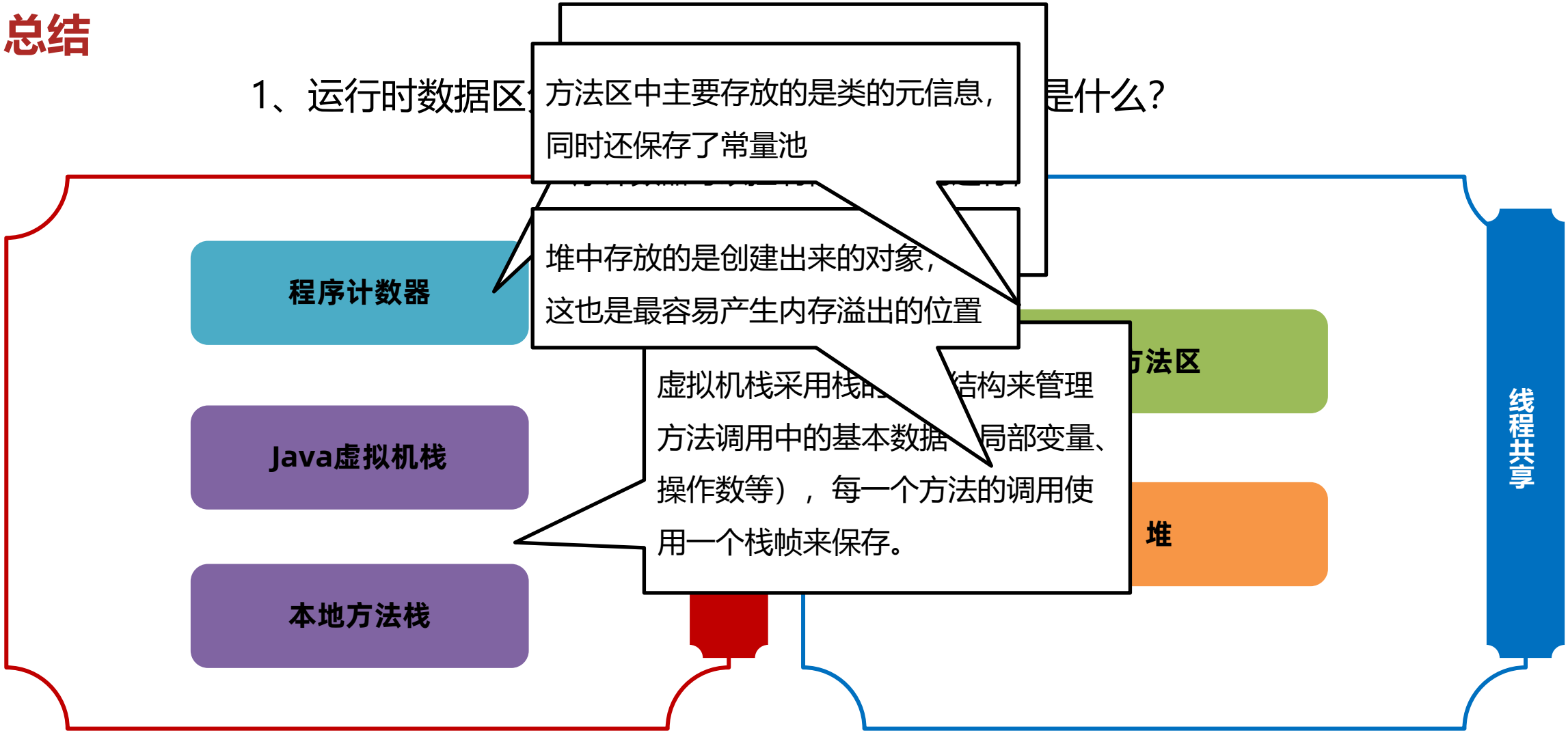
1、运行时数据区分成哪几部分，每一部分的作用是什么？

2、不同JDK版本之间运行时数据区域的区别是什么？

总结

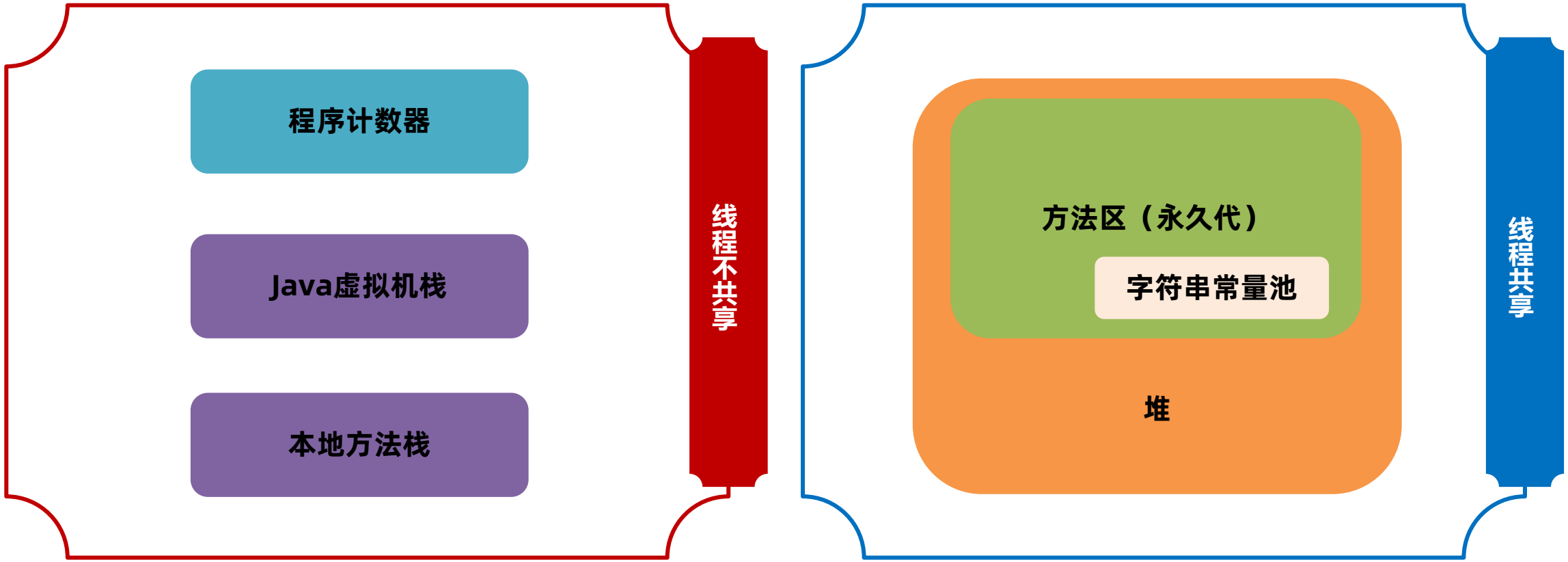
1、运行时数据区

是什么？



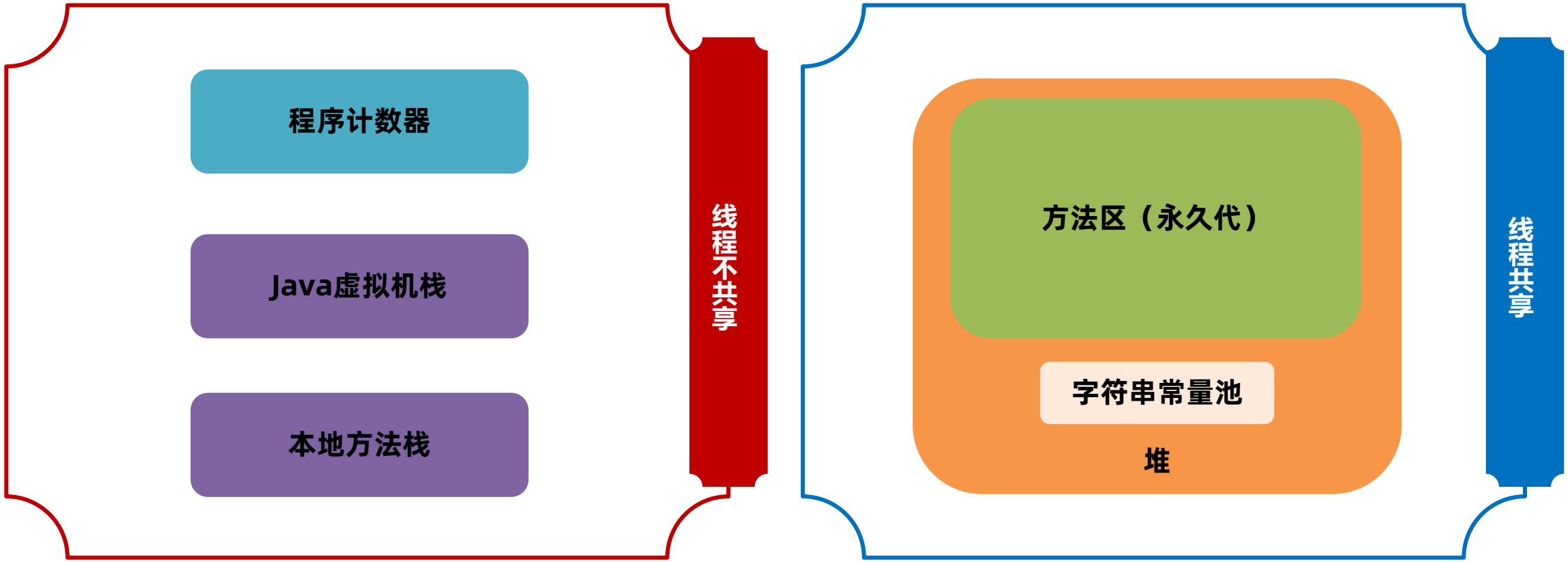
总结

2、不同JDK版本之间运行时数据区域的区别是什么？ JDK6



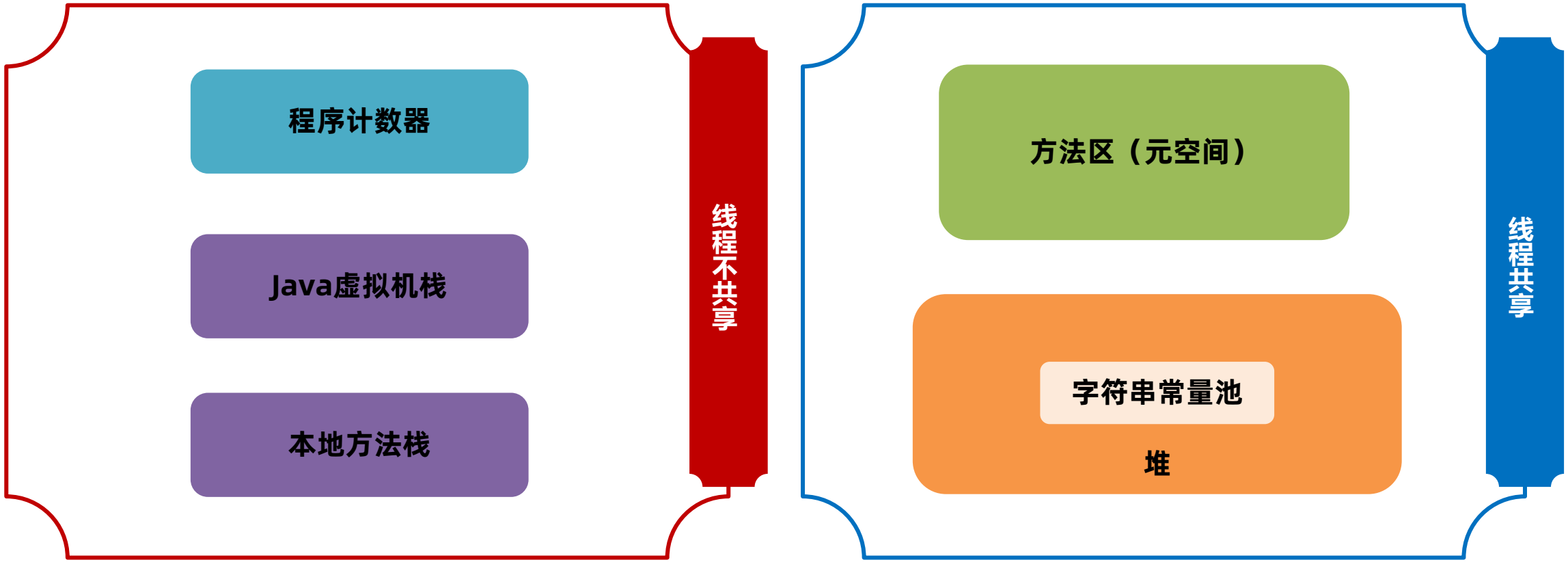
总结

2、不同JDK版本之间运行时数据区域的区别是什么？ JDK7



总结

2、不同JDK版本之间运行时数据区域的区别是什么？ JDK8



传智教育旗下高端IT教育品牌