

极客时间 Java 进阶训练营

第1课

JVM 核心技术--基础知识



KimmKing

Apache Dubbo/ShardingSphere PMC

个人介绍

Apache Dubbo/ShardingSphere PMC

前某集团高级技术总监/阿里架构师/某银行北京研发中心负责人

阿里云 MVP、腾讯 TVP、TGO 会员

10 多年研发管理和架构经验

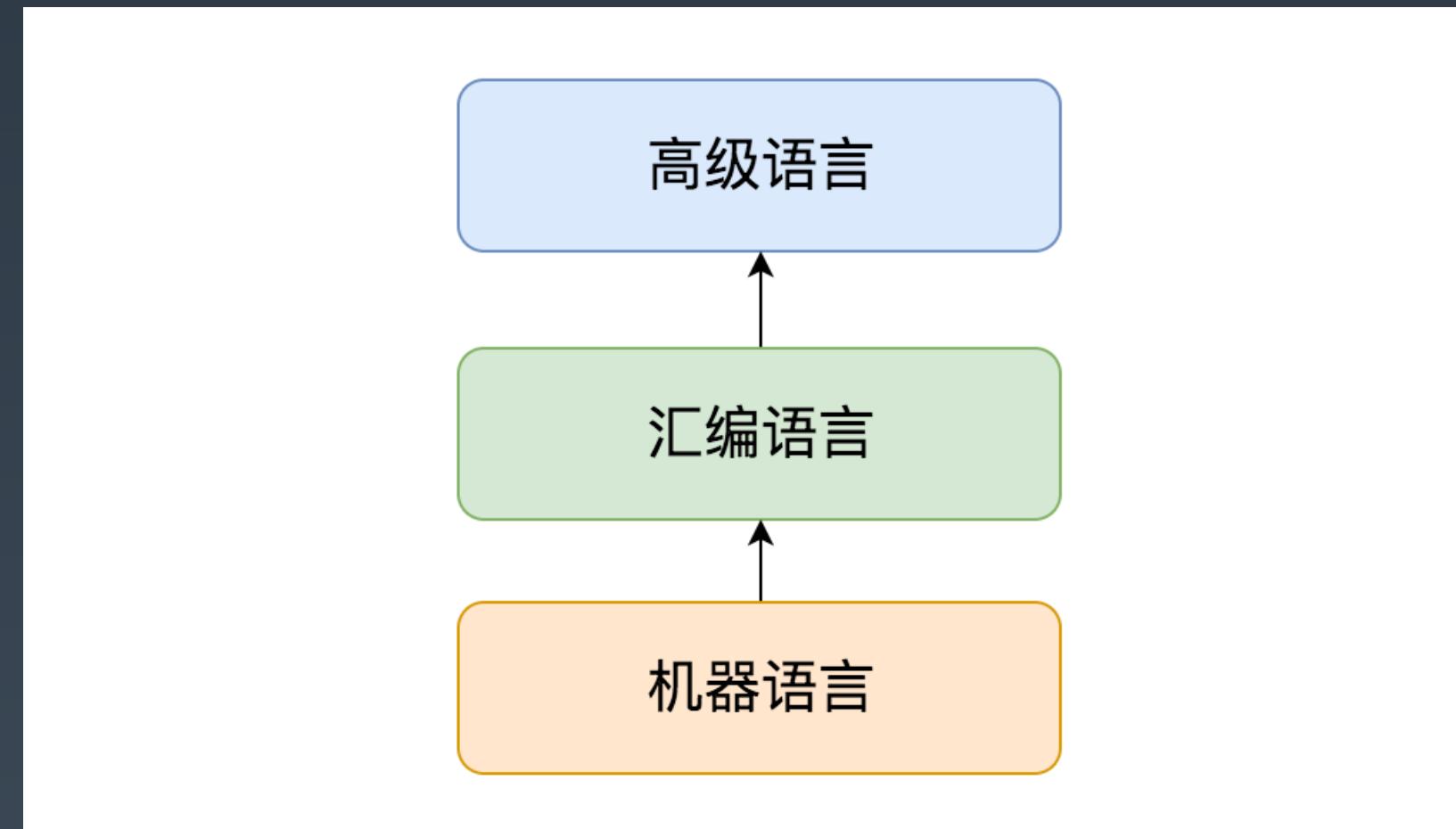
熟悉海量并发低延迟交易系统的设计实现

目录

1. JVM 基础知识：不积跬步，无以至千里
2. Java 字节码技术：不积细流，无以成江河
3. JVM 类加载器^{*}：山不辞土，故能成其高
4. JVM 内存模型^{*}：海不辞水，故能成其深
5. JVM 启动参数：博观而约取、厚积而薄发
6. 第1课总结回顾与作业实践

1. JVM 核心技术--基础知识

编程语言

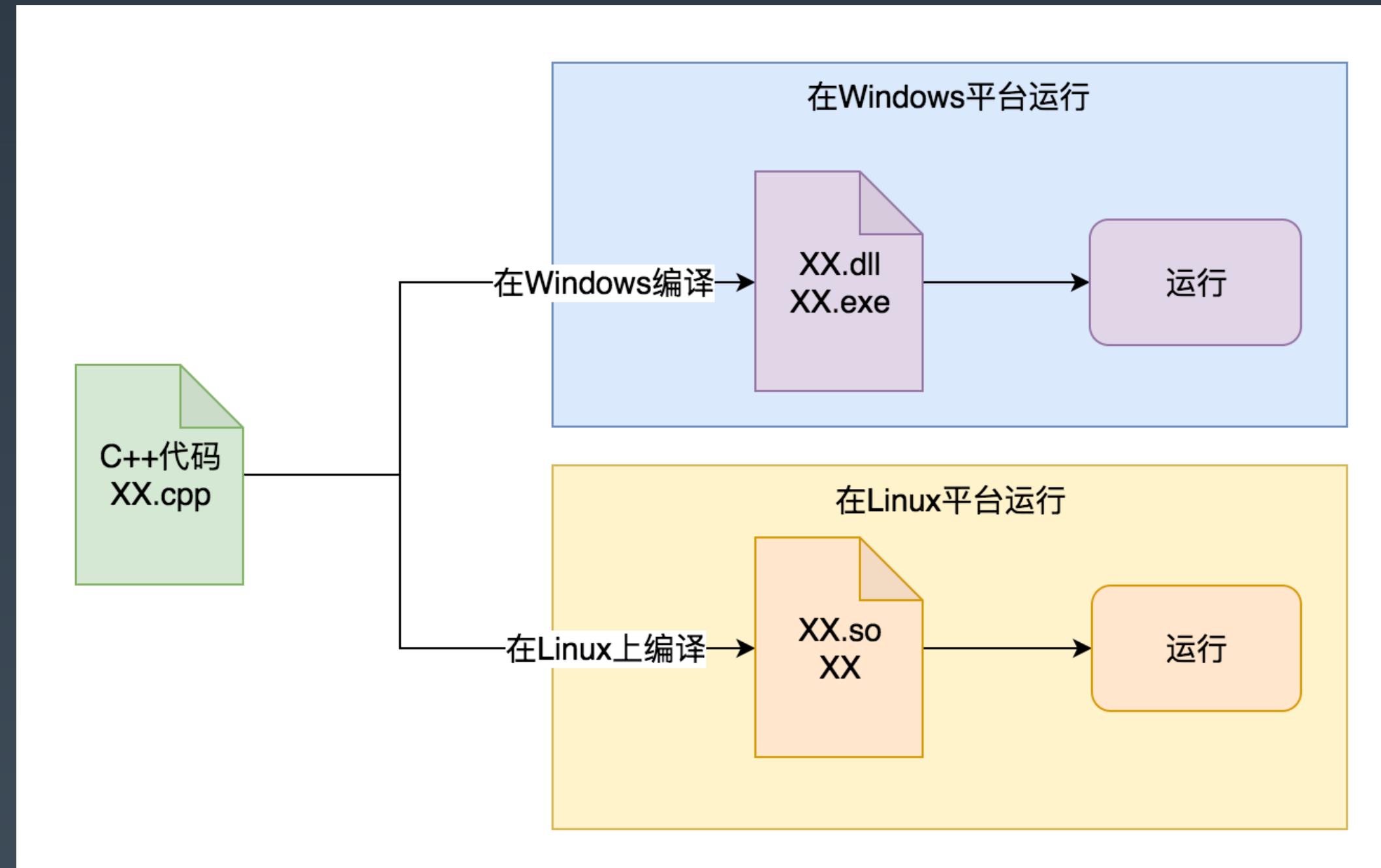


- 面向过程、面向对象、面向函数
- 静态类型、动态类型
- 编译执行、解释执行
- 有虚拟机、无虚拟机
- 有 GC、无 GC

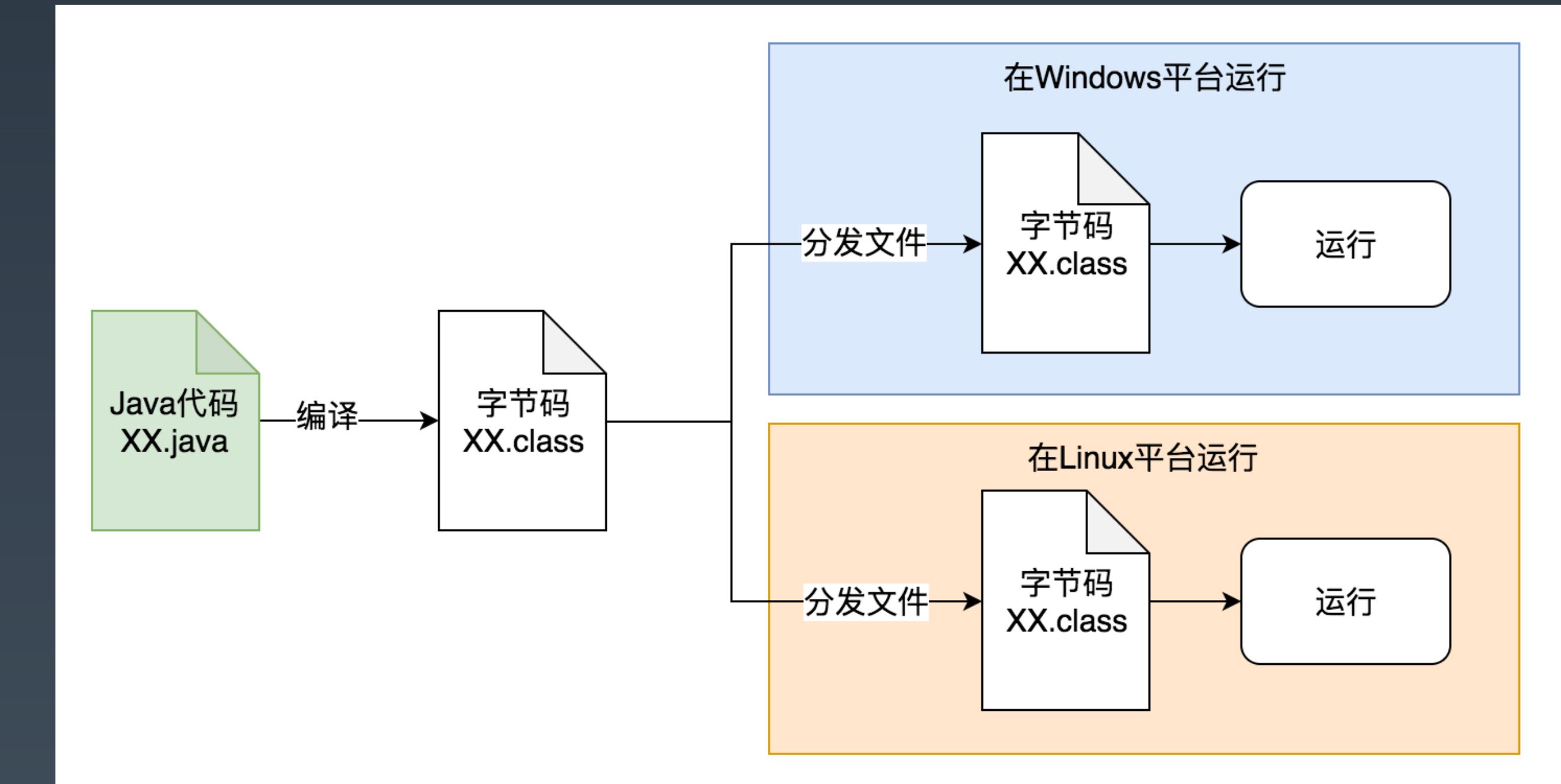


Java 是一种面向对象、静态类型、编译执行，
有 VM/GC 和运行时、跨平台的高级语言。

编程语言跨平台



源代码跨平台



二进制跨平台

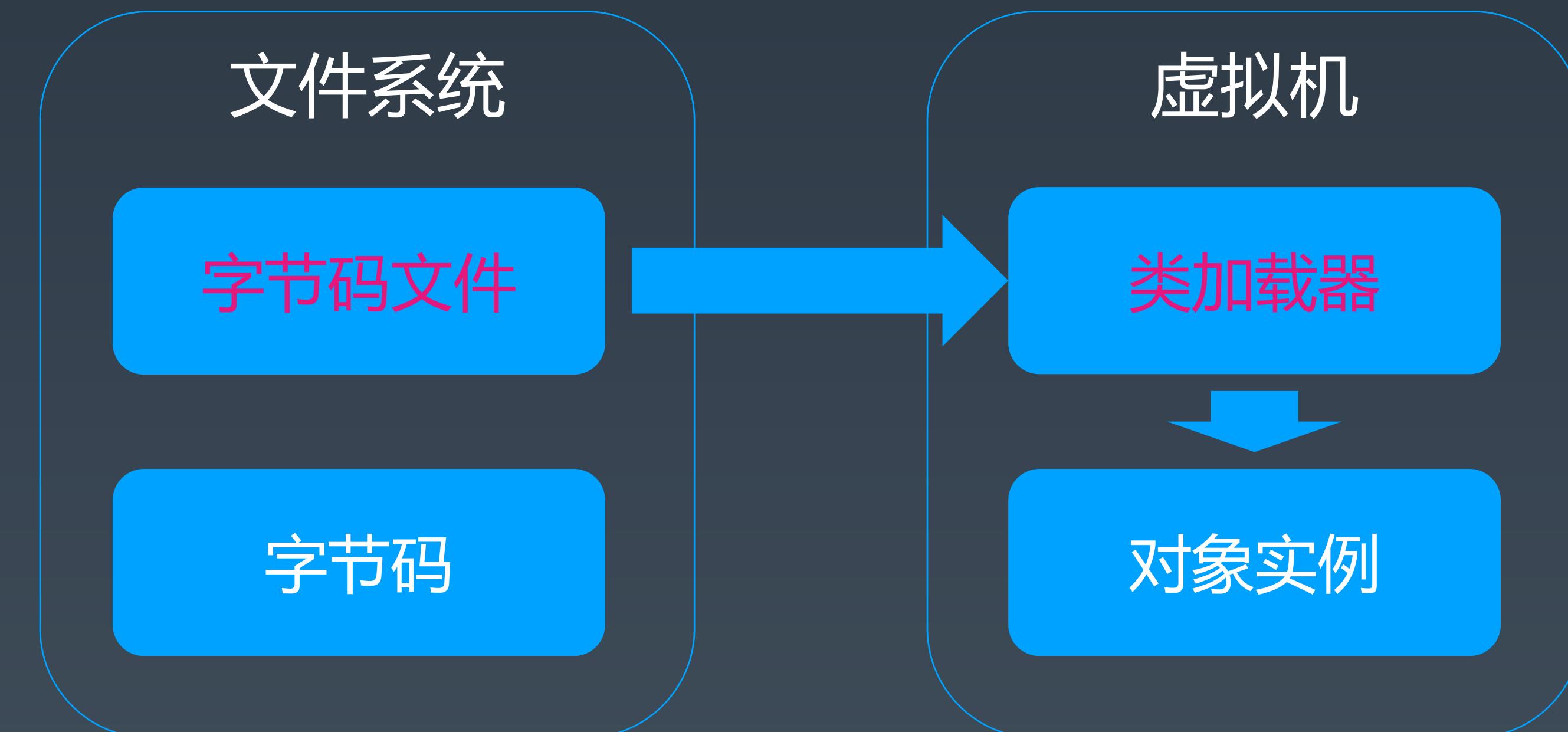
Java、C++、Rust 的区别

C/C++ 完全相信而且惯着程序员，让大家自行管理内存，可以编写很自由的代码，但一不小心就会造成内存泄漏等问题，导致程序崩溃。

Java/Golang 完全不相信程序员，但也惯着程序员。所有的内存生命周期都由 JVM 运行时统一管理。在绝大部分场景下，你可以非常自由地写代码，而且不用关心内存到底是什么情况。内存使用有问题的时候，我们可以通过 JVM 来进行信息相关的分析诊断和调整。这也是本课程的目标。

Rust 语言选择既不相信程序员，也不惯着程序员。让你在写代码的时候，必须清楚如何用 Rust 的规则管理好你的变量，好让机器能明白高效地分析和管理内存。但是这样会导致代码不利于人的理解，写代码很不自由，学习成本也很高。

字节码、类加载器、虚拟机



2. Java 字节码技术

什么是字节码？

Java bytecode 由单字节（byte）的指令组成，理论上最多支持 256 个操作码（opcode）。实际上 Java 只使用了200左右的操作码，还有一些操作码则保留给调试操作。

根据指令的性质，主要分为四个大类：

1. 栈操作指令，包括与局部变量交互的指令
2. 程序流程控制指令
3. 对象操作指令，包括方法调用指令
4. 算术运算以及类型转换指令

生成字节码

假如有一个最简单的类，源代码如下：

```
package demo.jvm0104;

public class HelloByteCode {
    public static void main(String[] args) {
        HelloByteCode obj = new HelloByteCode();
    }
}
```

编译：javac demo/jvm0104/HelloByteCode.java

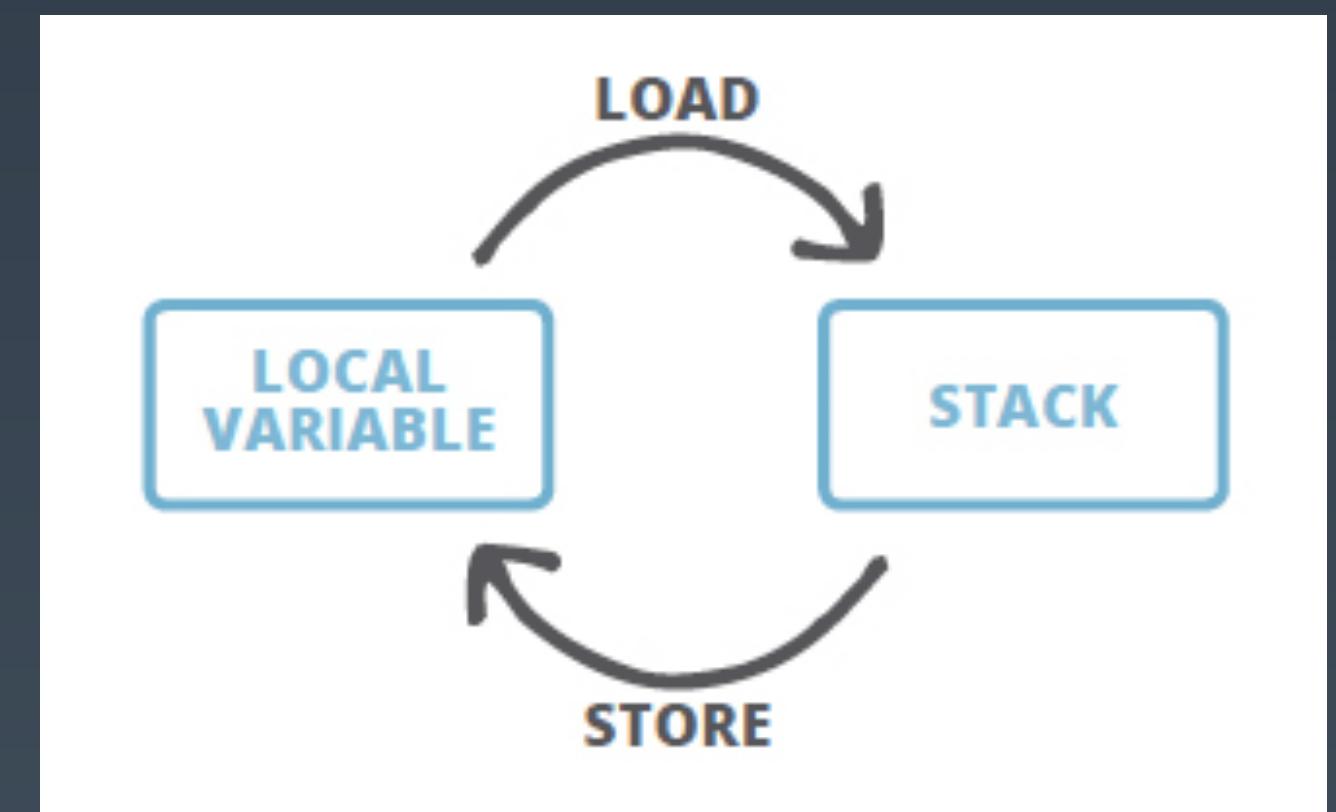
查看字节码：javap -c demo.jvm0104.HelloByteCode

最简单的字节码

结果如下所示：

```
Compiled from "HelloByteCode.java"
public class demo.jvm0104.HelloByteCode {
    public demo.jvm0104.HelloByteCode();
        Code:
            0: aload_0
            1: invokespecial #1                  // Method java/lang/Object."<
            4: return

    public static void main(java.lang.String[]);
        Code:
            0: new           #2                  // class demo/jvm0104/HelloBy
            3: dup
            4: invokespecial #3                  // Method "<init>":()V
            7: astore_1
            8: return
}
```



复杂点的例子

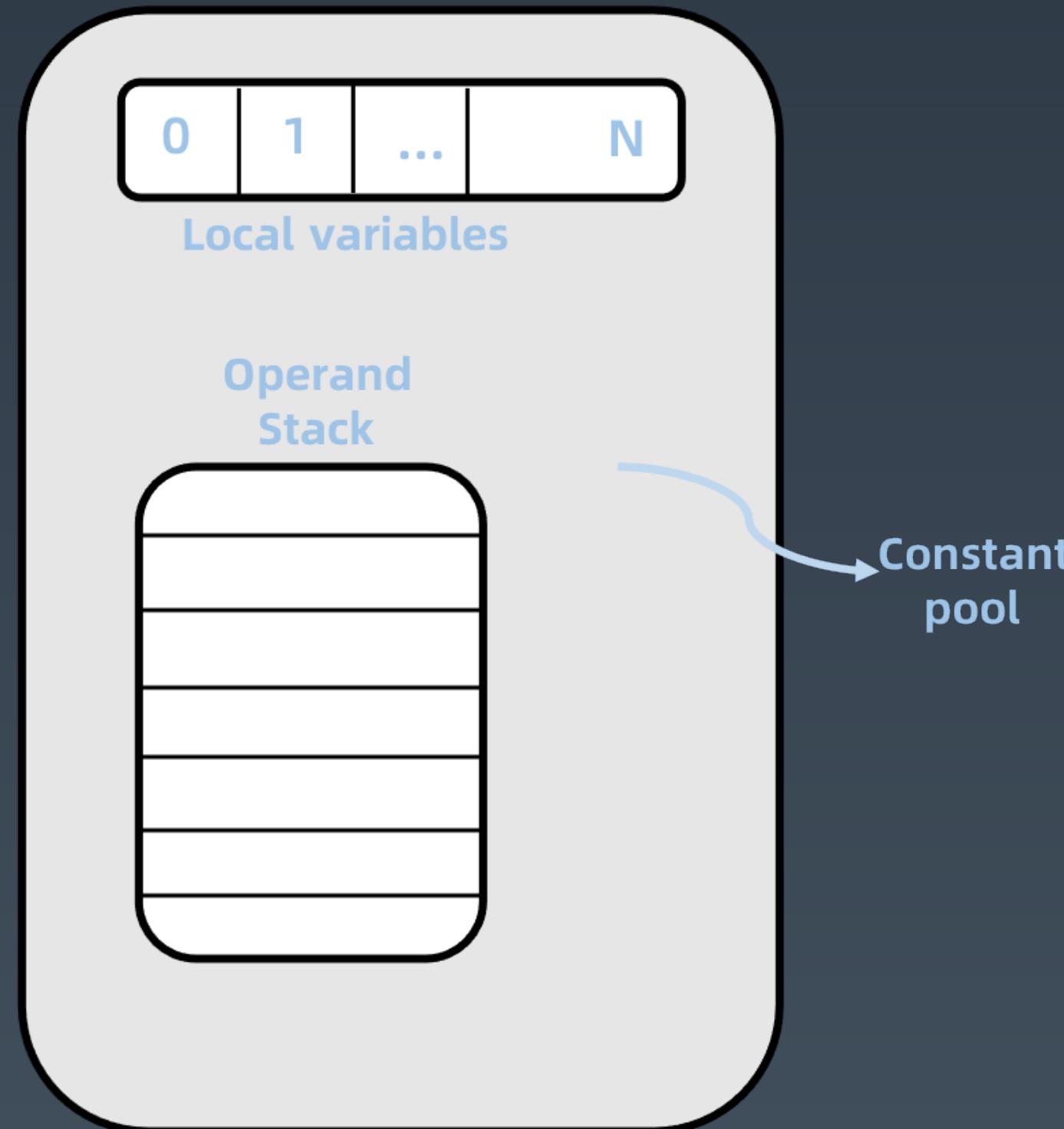
进一步: javap -c -verbose demo.jvm0104.HelloByteCode

```
Classfile /XXXXXXXX/demo/jvm0104/HelloByteCode.class
Last modified 2019-11-28; size 301 bytes
MD5 checksum 542cb70faf8b2b512a023e1a8e6c1308
Compiled from "HelloByteCode.java"
public class demo.jvm0104.HelloByteCode
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
#1 = Methodref #4.#13 // java/lang/Object."<init>":()V
#2 = Class #14 // demo/jvm0104/HelloByteCode
#3 = Methodref #2.#13 // demo/jvm0104/HelloByteCode."<init>":()V
#4 = Class #15 // java/lang/Object
#5 = Utf8 <init>
#6 = Utf8 ()V
#7 = Utf8 Code
#8 = Utf8 LineNumberTable
#9 = Utf8 main
#10 = Utf8 ([Ljava/lang/String;)V
#11 = Utf8 SourceFile
#12 = Utf8 HelloByteCode.java
#13 = NameAndType #5:#6 // "<init>":()V
#14 = Utf8 demo/jvm0104/HelloByteCode
#15 = Utf8 java/lang/Object
```

```
{
  public demo.jvm0104.HelloByteCode();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
      0: aload_0
      1: invokespecial #1 // Method java/lang/Object."<init>":()V
      4: return
  LineNumberTable:
    line 3: 0

  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 #2 // class demo/jvm0104/HelloByteCode
      3: dup
      4: invokespecial #3 // Method "<init>":()V
      7: astore_1
      8: return
  LineNumberTable:
    line 5: 0
    line 6: 8
}
SourceFile: "HelloByteCode.java"
```

字节码的运行时结构



JVM 是一台基于栈的计算机。

每个线程都有一个独属于自己的线程栈（JVM Stack），用于存储栈帧（Frame）。

每一次方法调用、JVM 都会自动创建一个栈帧。

栈帧由操作数栈、局部变量数组以及一个 Class 引用组成。

Class 引用指向当前方法在运行时常量池中对应的 Class。

从助记符到二进制

```
public static void main(java.lang.String[]);
Code:
0: new           #2                  // class demo/jvm0104/HelloBy
3: dup
4: invokespecial #3                // Method "<init>":()V
7: astore_1
8: return
```

0	1	2	3	4	5	6	7	8
new	00	02	dup	invoke special	00	03	astore_1	return

0	1	2	3	4	5	6	7	8
bb	00	02	59	b7	00	03	4c	b1

```
5 0000040 6c 65 01 00 04 6d 61 69 6e 01 00 16 28 5b
6 0000050 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e
7 0000060 29 56 01 00 0a 53 6f 75 72 63 65 46 69 6c 65 01 | )V...SourceFile.
8 0000070 00 12 48 65 6c 6c 6f 42 79 74 65 43 6f 64 65 2e | ..HelloByteCode.
9 0000080 6a 61 76 61 0c 00 05 00 06 01 00 1a 64 65 6d 6f | java.....demo
10 0000090 2f 6a 76 6d 30 31 30 34 2f 48 65 6c 6c 6f 42 79 | /jvm0104/HelloBy
11 00000a0 74 65 43 6f 64 65 01 00 10 6a 61 76 61 2f 6c 61 | teCode...java/la
12 00000b0 6e 67 2f 4f 62 6a 65 63 74 00 21 00 02 00 04 00 | ng/Object.!....
13 00000c0 00 00 00 02 00 01 00 05 00 06 00 01 00 07 00 | .....
14 00000d0 00 00 1d 00 01 00 01 00 00 00 05 2a b7 00 01 b1 | .....*...
15 00000e0 00 00 00 01 00 08 00 00 00 06 00 01 00 00 00 03 | .....
16 00000f0 00 09 00 09 00 0a 00 01 00 07 00 00 00 25 00 02 | .....%..
17 0000100 00 02 00 00 00 09 bb 00 02 59 b7 00 03 4c b1 00 | .....Y...L...
18 0000110 00 00 01 00 08 00 00 00 0a 00 02 00 00 00 05 00 | .....
19 0000120 08 00 06 00 01 00 0b 00 00 00 02 00 00 0c | .....
20 000012d
```

四则运行的例子

```
package demo.jvm0104;
//移动平均数
public class MovingAverage {
    private int count = 0;
    private double sum = 0.0D;
    public void submit(double value){
        this.count++;
        this.sum += value;
    }
    public double getAvg(){
        if(0 == this.count){ return sum; }
        return this.sum/this.count;
    }
}
```

```
package demo.jvm0104;
public class LocalVariableTest {
    public static void main(String[] args) {
        MovingAverage ma = new MovingAverage();
        int num1 = 1;
        int num2 = 2;
        ma.submit(num1);
        ma.submit(num2);
        double avg = ma.getAvg();
    }
}
```

数值处理与本地变量表

```
0: new           #2                  // class demo/jvm0104/MovingAverage
3: dup
4: invokespecial #3                // Method demo/jvm0104/MovingAverage.<init>
7: astore_1
8: iconst_1
9: istore_2
10: iconst_2
11: istore_3
12: aload_1
13: iload_2
14: i2d
15: invokevirtual #4               // Method demo/jvm0104/MovingAverage.average
18: aload_1
19: iload_3
20: i2d
21: invokevirtual #4               // Method demo/jvm0104/MovingAverage.average
24: aload_1
25: invokevirtual #5               // Method demo/jvm0104/MovingAverage.average
28: dstore        4
30: return
```

LineNumberTable:

line 5: 0
line 6: 8
line 7: 10
line 8: 12
line 9: 18
line 10: 24
line 11: 30

LocalVariableTable:

Start	Length	Slot	Name	Signature
0	31	0	args	[Ljava/lang/String;
8	23	1	ma	Ldemo/jvm0104/MovingAverage;
10	21	2	num1	I
12	19	3	num2	I
30	1	4	avg	D

算数操作与类型转换

算数操作与类型转换

	add +	sub -	mult *	divide /	remainder %	negate -()
int	iadd	isub	imul	idiv	irem	ineg
long	ladd	lsub	lmul	ldiv	lrem	lneg
float	fadd	fsub	fmul	fdiv	frem	fneg
double	dadd	dsub	dmul	ddiv	drem	dneg

To

	int	long	float	double	byte	char	short
int	-	i2l	i2f	i2d	i2b	i2c	i2s
long	l2i	-	l2f	l2d	-	-	-
float	f2i	f2l	-	f2d	-	-	-
double	d2i	d2l	d2f	-	-	-	-

Form

一个完整的循环控制

```
package demo.jvm0104;
public class ForLoopTest {
    private static int[] numbers = {1, 6, 8};
    public static void main(String[] args) {
        MovingAverage ma = new MovingAverage();
        for (int number : numbers) {
            ma.submit(number);
        }
        double avg = ma.getAvg();
    }
}
```

```
0: new          #2           // class
3: dup
4: invokespecial #3           // Method
7: astore_1
8: getstatic     #4           // Field
11: astore_2
12: aload_2
13: arraylength
14: istore_3
15: iconst_0
16: istore      4
18: iload       4
20: iload_3
21: if_icmpge   43
24: aload_2
25: iload
27: iaload
28: istore      5
30: aload_1
31: iload       5
33: i2d
34: invokevirtual #5           // Method
37: iinc        4, 1
40: goto         18
43: aload_1
44: invokevirtual #6           // Method
47: dstore_2
48: return
```

方法调用的指令

Invokestatic: 顾名思义，这个指令用于调用某个类的静态方法，这是方法调用指令中最快的一个。

Invokespecial : 用来调用构造函数，但也可以用于调用同一个类中的 private 方法，以及可见的超类方法。

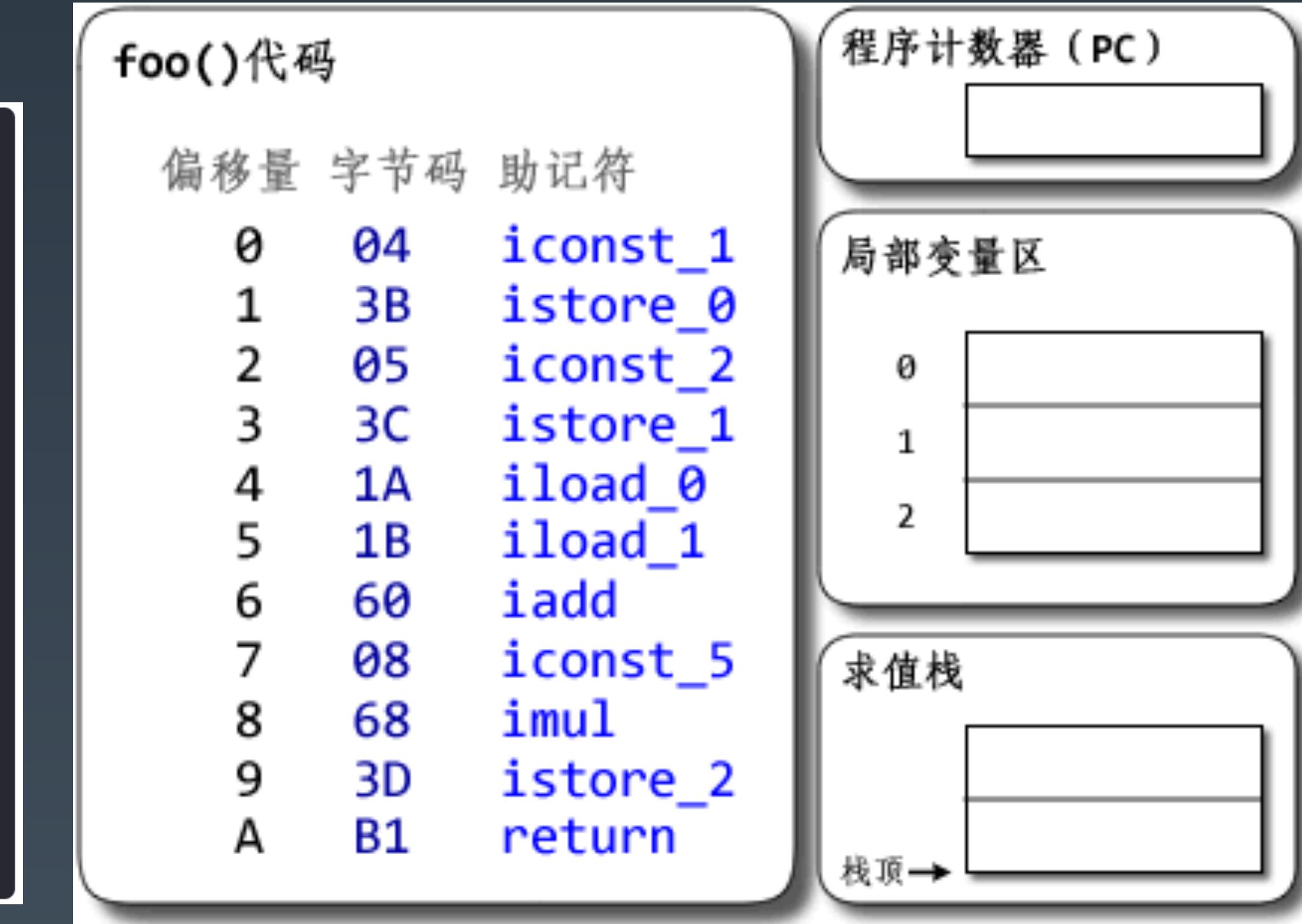
invokevirtual : 如果是具体类型的目标对象，`invokevirtual` 用于调用公共、受保护和 package 级的私有方法。

invokeinterface : 当通过接口引用来调用方法时，将会编译为 `invokeinterface` 指令。

invokedynamic : JDK7 新增加的指令，是实现“动态类型语言”（Dynamically Typed Language）支持而进行的升级改进，同时也是 JDK8 以后支持 lambda 表达式的实现基础。

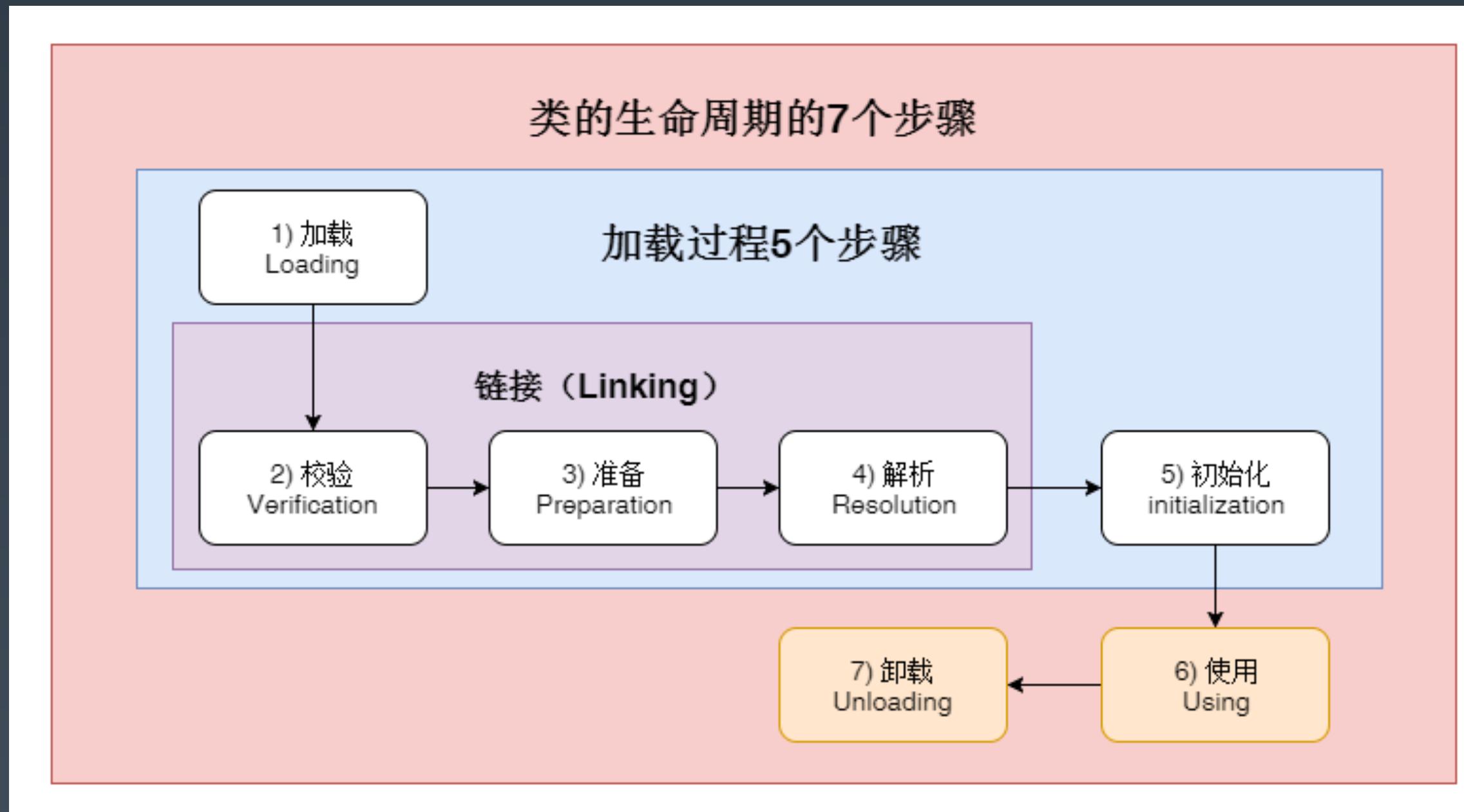
一个动态例子

```
public class Demo {  
  
    public static void foo() {  
  
        int a = 1;  
  
        int b = 2;  
  
        int c = (a + b) * 5;  
    }  
}
```



3. JVM 类加载器

类的生命周期



1. 加载 (Loading) : 找 Class 文件
2. 验证 (Verification) : 验证格式、依赖
3. 准备 (Preparation) : 静态字段、方法表
4. 解析 (Resolution) : 符号解析为引用
5. 初始化 (Initialization) : 构造器、静态变量赋值、静态代码块
6. 使用 (Using)
7. 卸载 (Unloading)

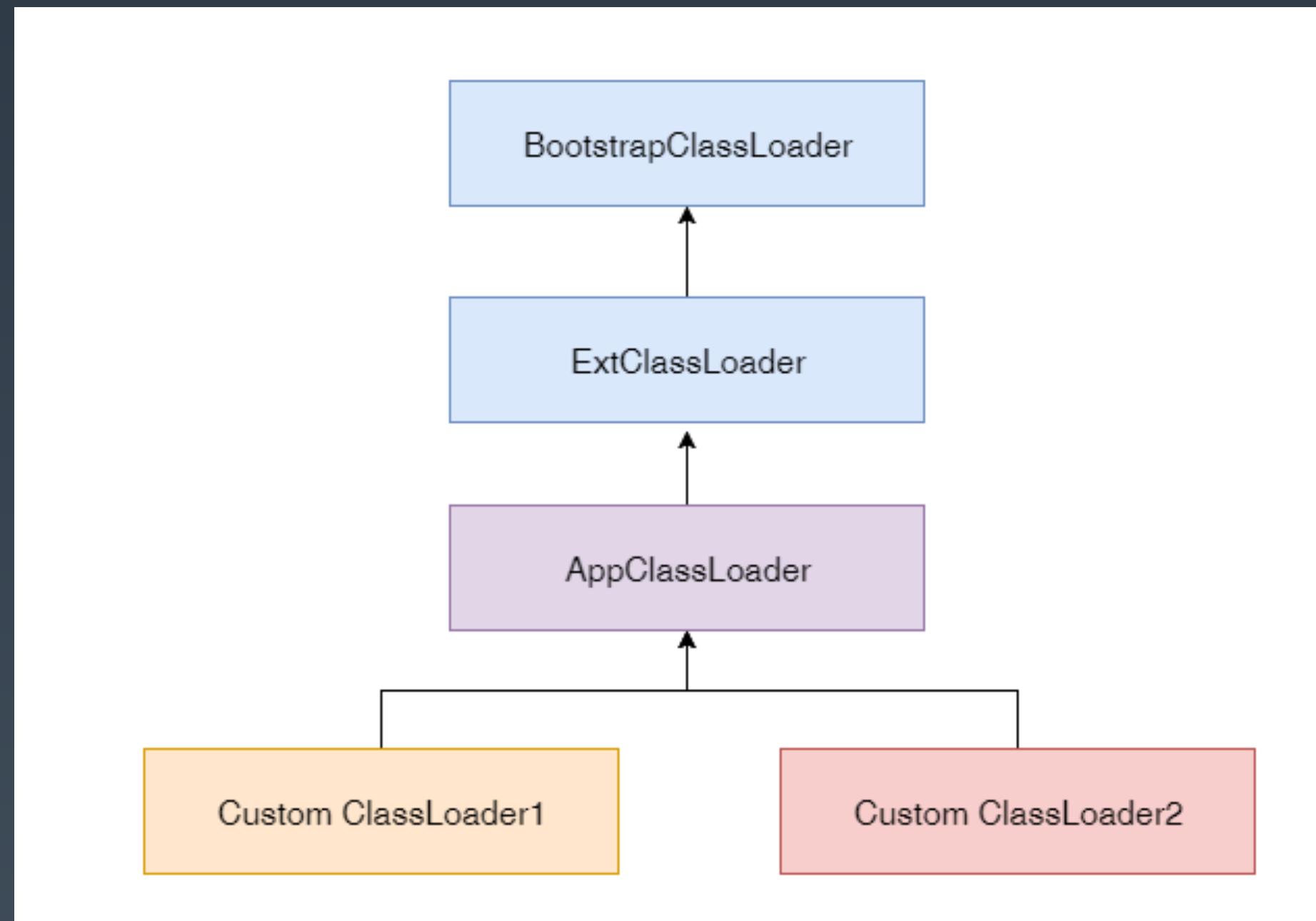
类的加载时机

1. 当虚拟机启动时，初始化用户指定的主类，就是启动执行的 main 方法所在的类；
2. 当遇到用以新建目标类实例的 new 指令时，初始化 new 指令的目标类，就是 new 一个类的时候要初始化；
3. 当遇到调用静态方法的指令时，初始化该静态方法所在的类；
4. 当遇到访问静态字段的指令时，初始化该静态字段所在的类；
5. 子类的初始化会触发父类的初始化；
6. 如果一个接口定义了 default 方法，那么直接实现或者间接实现该接口的类的初始化，会触发该接口的初始化；
7. 使用反射 API 对某个类进行反射调用时，初始化这个类，其实跟前面一样，反射调用要么是已经有实例了，要么是静态方法，都需要初始化；
8. 当初次调用 MethodHandle 实例时，初始化该 MethodHandle 指向的方法所在的类。

不会初始化（可能会加载）

1. 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
2. 定义对象数组，不会触发该类的初始化。
3. 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
4. 通过类名获取 Class 对象，不会触发类的初始化，`Hello.class` 不会让 Hello 类初始化。
5. 通过 `Class.forName` 加载指定类时，如果指定参数 `initialize` 为 `false` 时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
`(Class.forName("jvm.Hello"))` 默认会加载 Hello 类。
6. 通过 `ClassLoader` 默认的 `loadClass` 方法，也不会触发初始化动作（加载了，但是不初始化）。

三类加载器

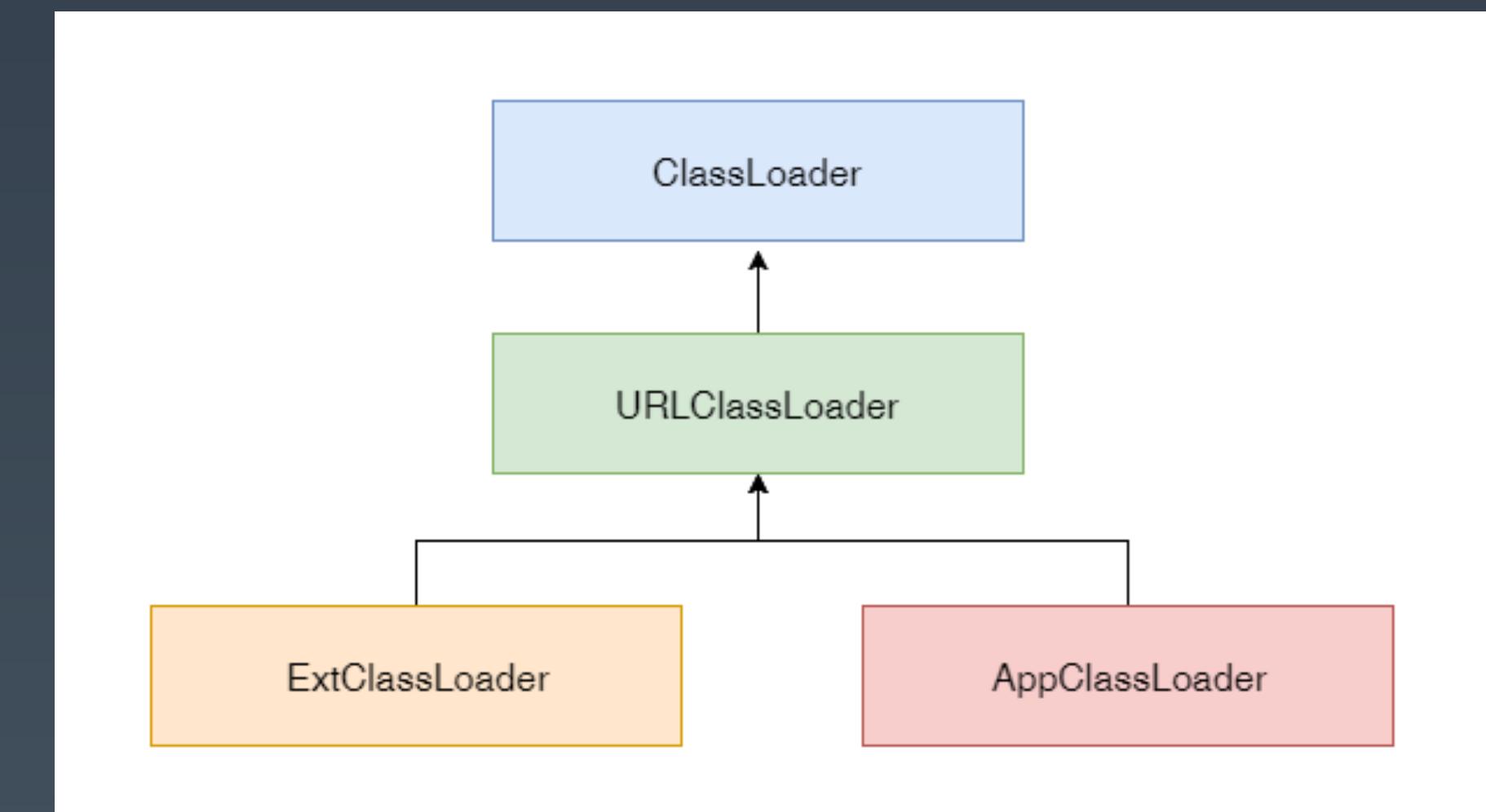


三类加载器：

1. 启动类加载器 (BootstrapClassLoader)
2. 扩展类加载器 (ExtClassLoader)
3. 应用类加载器 (AppClassLoader)

加载器特点：

1. 双亲委托
2. 负责依赖
3. 缓存加载



显示当前 ClassLoader 加载了哪些 Jar?

```
public class JvmClassLoaderPrintPath {
    public static void main(String[] args) {
        // 启动类加载器
        URL[] urls = sun.misc.Launcher.getBootstrapClassPath().getURLs();
        System.out.println("启动类加载器");
        for(URL url : urls) {
            System.out.println(" ==> " + url.toExternalForm());
        }

        // 扩展类加载器
        printClassLoader("扩展类加载器", JvmClassLoaderPrintPath.class.getClassLoader());
        // 应用类加载器
        printClassLoader("应用类加载器", JvmClassLoaderPrintPath.class.getClassLoader());
    }

    public static void printClassLoader(String name, ClassLoader CL){
        if(CL != null) {
            System.out.println(name + " ClassLoader -> " + CL.toString());
            printURLForClassLoader(CL);
        }else{
            System.out.println(name + " ClassLoader -> null");
        }
    }
}
```

```
public static void printURLForClassLoader(ClassLoader CL){
    Object ucp = insightField(CL, "ucp");
    Object path = insightField(ucp, "path");
    ArrayList ps = (ArrayList) path;
    for (Object p : ps){
        System.out.println(" ==> " + p.toString());
    }
}

private static Object insightField(Object obj, String fName) {
    try {
        Field f = null;
        if(obj instanceof URLClassLoader){
            f = URLClassLoader.class.getDeclaredField(fName);
        }else{
            f = obj.getClass().getDeclaredField(fName);
        }
        f.setAccessible(true);
        return f.get(obj);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
}
```

启动类加载器

```
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/resources.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/rt.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/sunrsasign.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/jsse.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/jce.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/charsets.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/jfr.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/classes
```

扩展类加载器 ClassLoader -> sun.misc.Launcher\$ExtClassLoader@15db9742

```
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/access-bridge.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/cldrdata.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/dnsns.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/jaccess.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/jfxrt.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/locatedata.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/nashorn.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/sunec.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/sunjce_provider.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/sunmscapi.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/sunpkcs11.jar
==> file:/D:/Program%20Files/Java/jdk1.8.0_231/jre/lib/ext/zipfs.jar
```

应用类加载器 ClassLoader -> sun.misc.Launcher\$AppClassLoader@73d16e93

```
==> file:/D:/git/studyjava/build/classes/java/main/
==> file:/D:/git/studyjava/build/resources/main
```

自定义 ClassLoader

```
package jvm;

public class Hello {
    static {
        System.out.println("Hello Class Initialized!");
    }
}
```

```
$ java jvm.HelloClassLoader
Hello Class Initialized!
```

```
package jvm;

import java.util.Base64;

public class HelloClassLoader extends ClassLoader {

    public static void main(String[] args) {
        try {
            new HelloClassLoader().findClass("jvm.Hello").newInstance();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        }
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        String helloBase64 = "yv66vgAAADQAHwoABgARcqASABMIABQKABUAFgcAFwcJ...";  
"hbFZhcmLhYmxlVGFibGUBAAR0aGlzAQALTGp2bS9IZWxsbszBAg8Y2xJ...";  
"GxvIENsYXNzIEluaXRpYwpxemVkiQcAHQ AeAQAJanZtL0hlbGxvA...";  
"YS9pb9QcmIudFN0cmVhbTsBABNqYXZhL2lvL1ByaW50U3RyZWftAQAH...";  
"ABAkAAAAvAAEAAQAAAQtwABsQAAAIAcGAAAAYAAQAAAAMACwAAA...";  
"AAAACgACAAAABgAIAAcAAQAPAAAAAgAQ";  
  
        byte[] bytes = decode(helloBase64);
        return defineClass(name,bytes,0,bytes.length);
    }

    public byte[] decode(String base64){
        return Base64.getDecoder().decode(base64);
    }
}
```

添加引用类的几种方式

- 1、放到 JDK 的 lib/ext 下，或者 -Djava.ext.dirs
- 2、java-cp/classpath 或者 class 文件放到当前路径
- 3、自定义 ClassLoader 加载
- 4、拿到当前执行类的 ClassLoader，反射调用 addUrl 方法添加 Jar 或路径（JDK9 无效）

```
package jvm;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

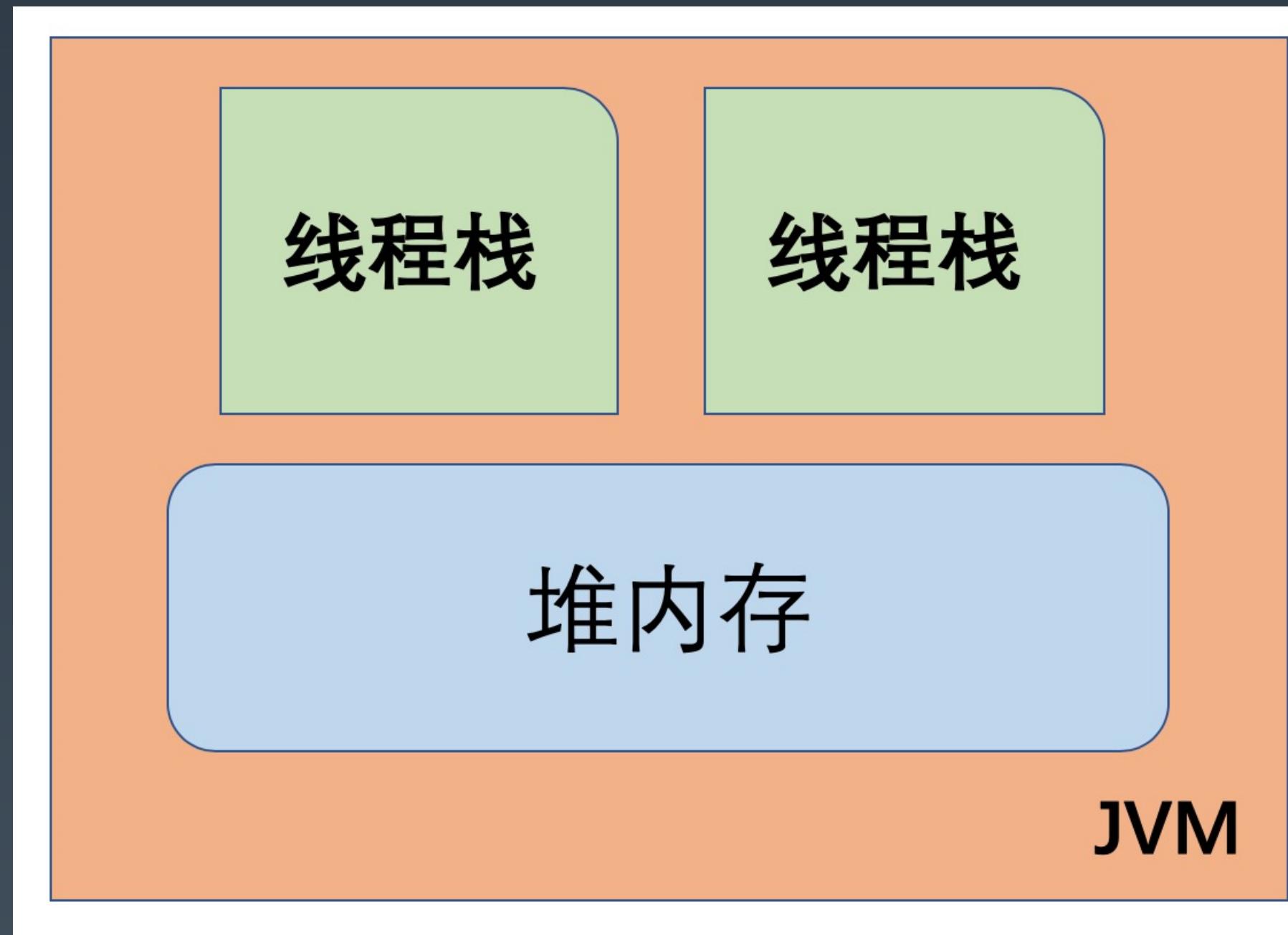
public class JvmAppClassLoaderAddURL {

    public static void main(String[] args) {

        String appPath = "file:/d:/app/";
        URLClassLoader urlClassLoader = (URLClassLoader) JvmAppClassLoaderAddURL.class.getClassLoader();
        try {
            Method addURL = URLClassLoader.class.getDeclaredMethod("addURL", URL.class);
            addURL.setAccessible(true);
            URL url = new URL(appPath);
            addURL.invoke(urlClassLoader, url);
            Class.forName("jvm.Hello"); // 效果跟Class.forName("jvm.Hello").newInstance()一样
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

4. JVM 内存模型

JVM 内存结构



每个线程都只能访问自己的线程栈。

每个线程都不能访问（看不见）其他线程的局部变量。

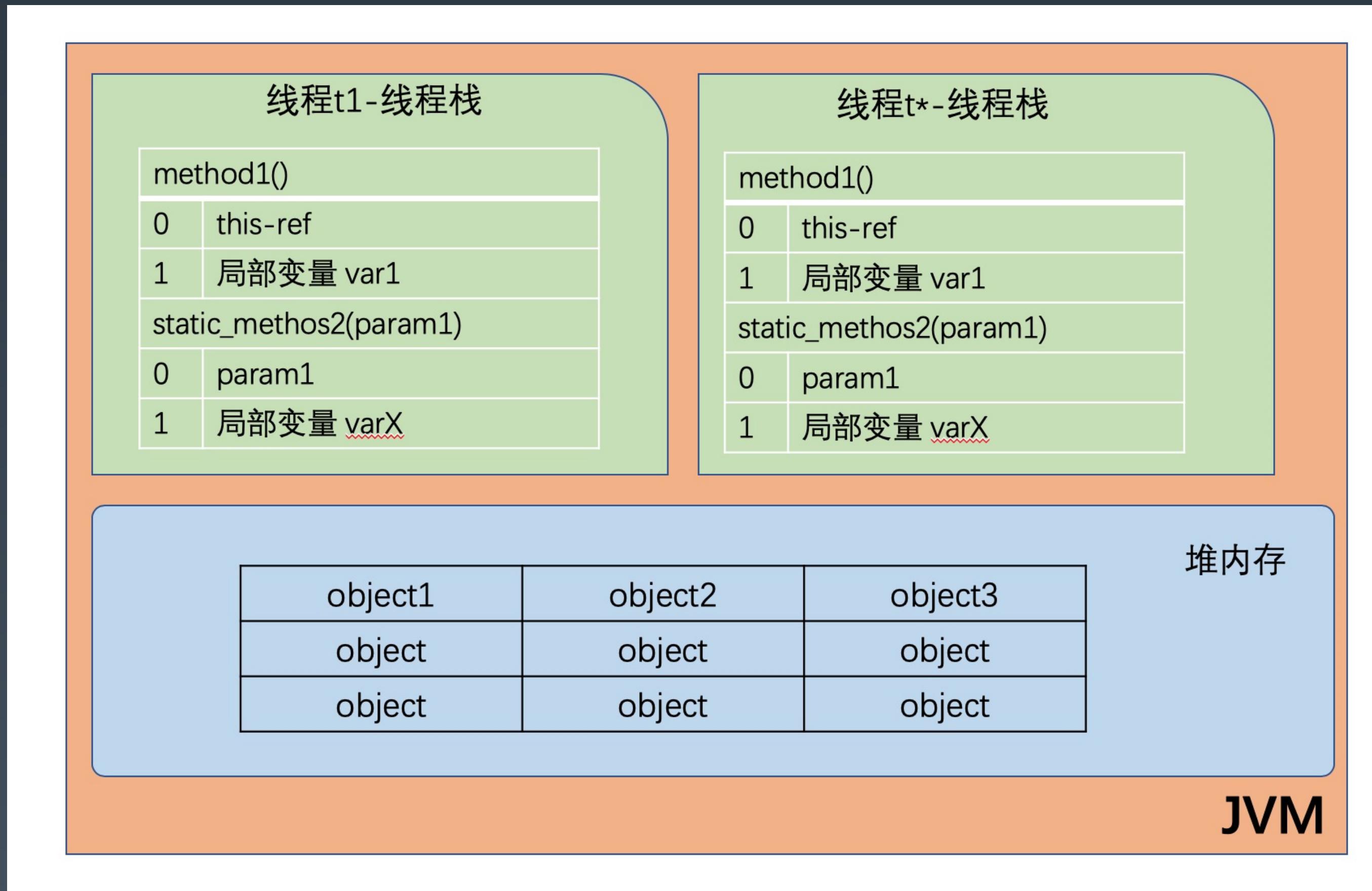
所有原生类型的局部变量都存储在线程栈中，因此对其他线程是不可见的。

线程可以将一个原生变量值的副本传给另一个线程，但不能共享原生局部变量本身。

堆内存中包含了 Java 代码中创建的所有对象，不管是哪个线程创建的。其中也涵盖了包装类型（例如 Byte, Integer, Long 等）。

不管是创建一个对象并将其赋值给局部变量，还是赋值给另一个对象的成员变量，创建的对象都会被保存到堆内存中。

JVM 内存结构



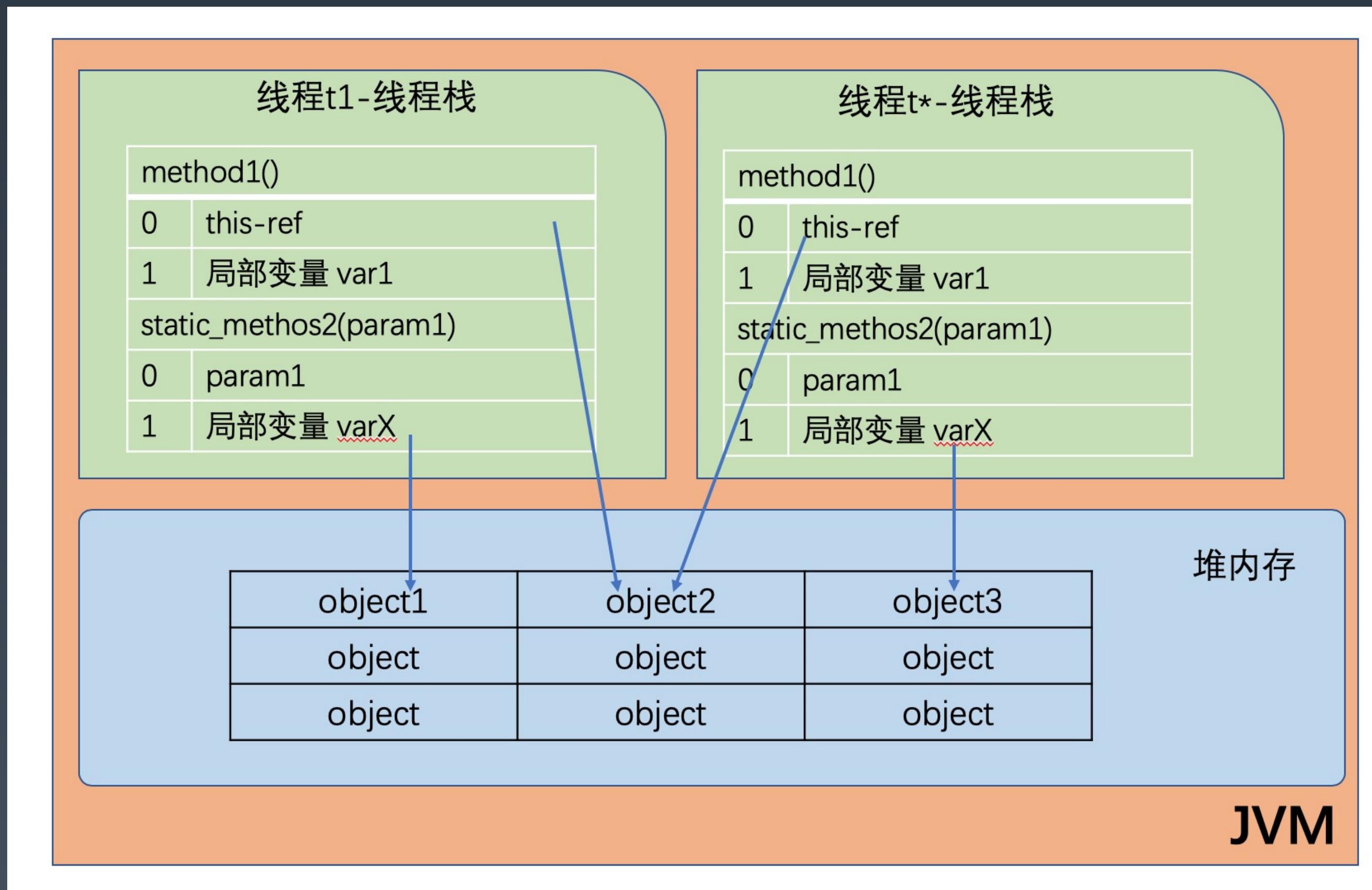
如果是原生数据类型的局部变量，那么它的内容就全部保留在线程栈上。

如果是对象引用，则栈中的局部变量槽位中保存着对象的引用地址，而实际的对象内容保存在堆中。

对象的成员变量与对象本身一起存储在堆上，不管成员变量的类型是原生数值，还是对象引用。

类的静态变量则和类定义一样都保存在堆中。

JVM 内存结构



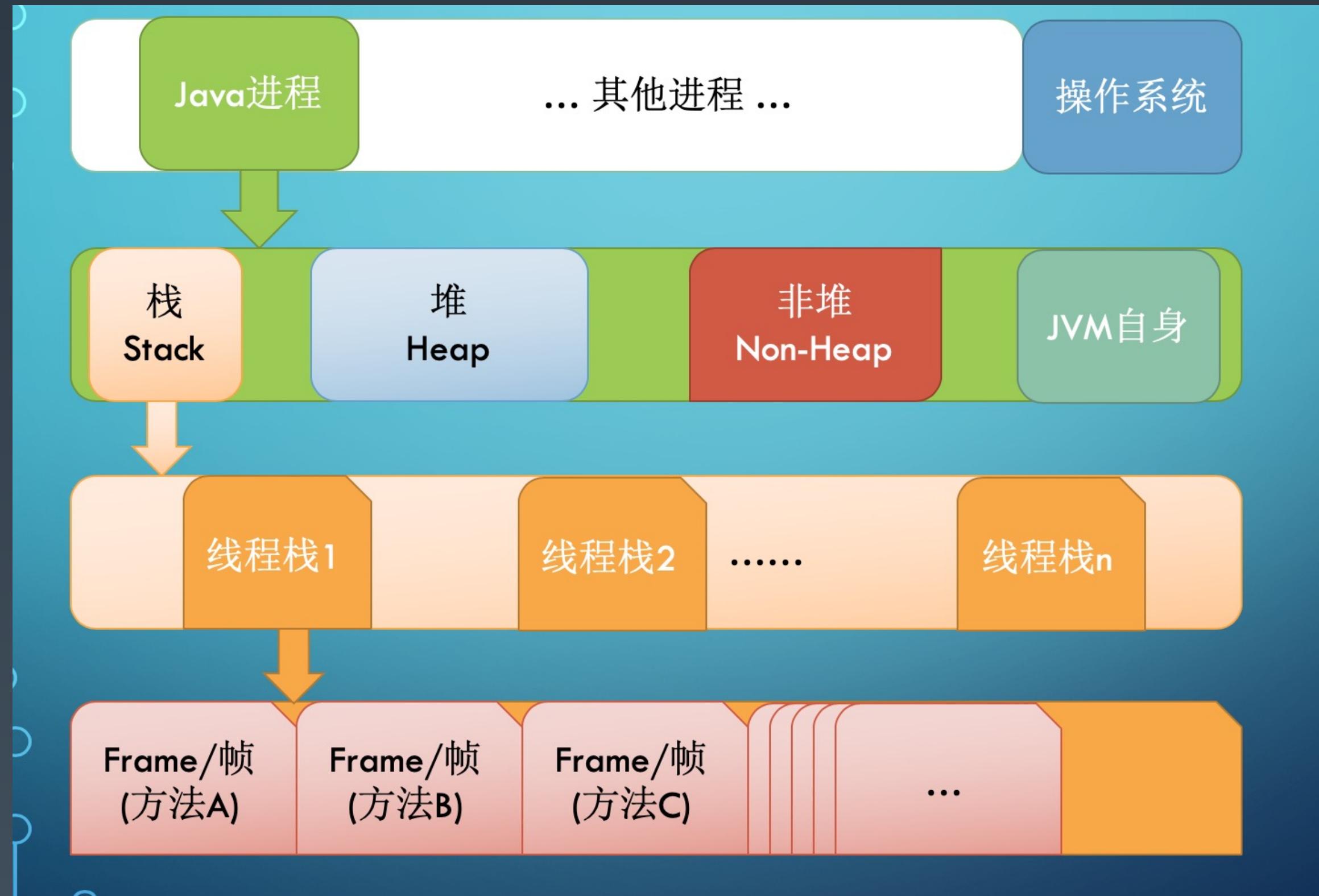
总结一下：方法中使用的原生数据类型和对象引用地址在栈上存储；对象、对象成员与类定义、静态变量在堆上。

堆内存又称为“共享堆”，堆中的所有对象，可以被所有线程访问，只要他们能拿到对象的引用地址。

如果一个线程可以访问某个对象时，也就可以访问该对象的成员变量。

如果两个线程同时调用某个对象的同一方法，则它们都可以访问到这个对象的成员变量，但每个线程的局部变量副本是独立的。

JVM 内存整体结构

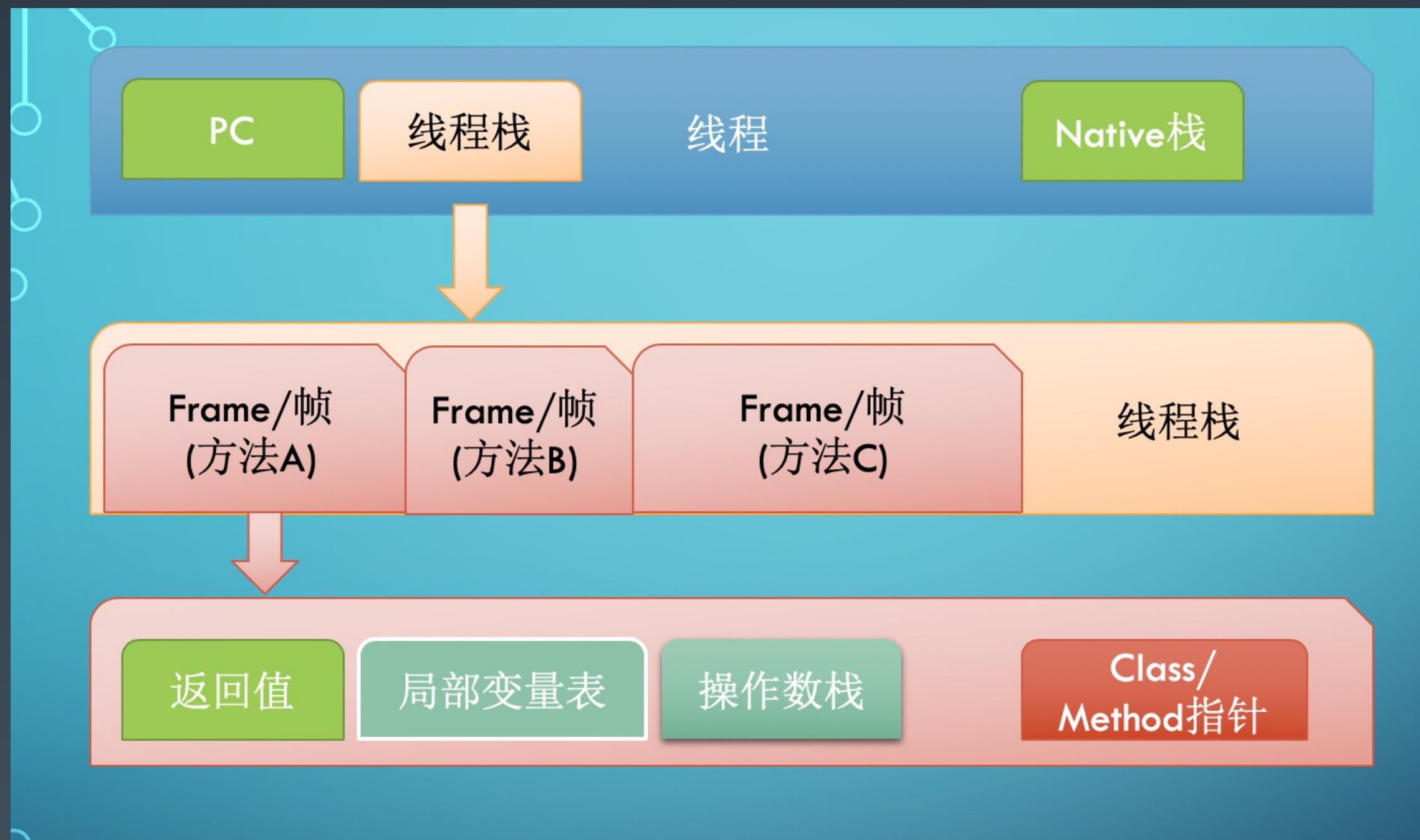


每启动一个线程，JVM 就会在栈空间栈分配对应的 线程栈, 比如 1MB 的空间 (-Xss1m)。

线程栈也叫做 Java 方法栈。如果使用了 JNI 方法，则会分配一个单独的本地方法栈 (Native Stack)。

线程执行过程中，一般会有多个方法组成调用栈 (Stack Trace)，比如 A 调用 B, B 调用 C...每执行到一个方法，就会创建对应的 栈帧 (Frame)。

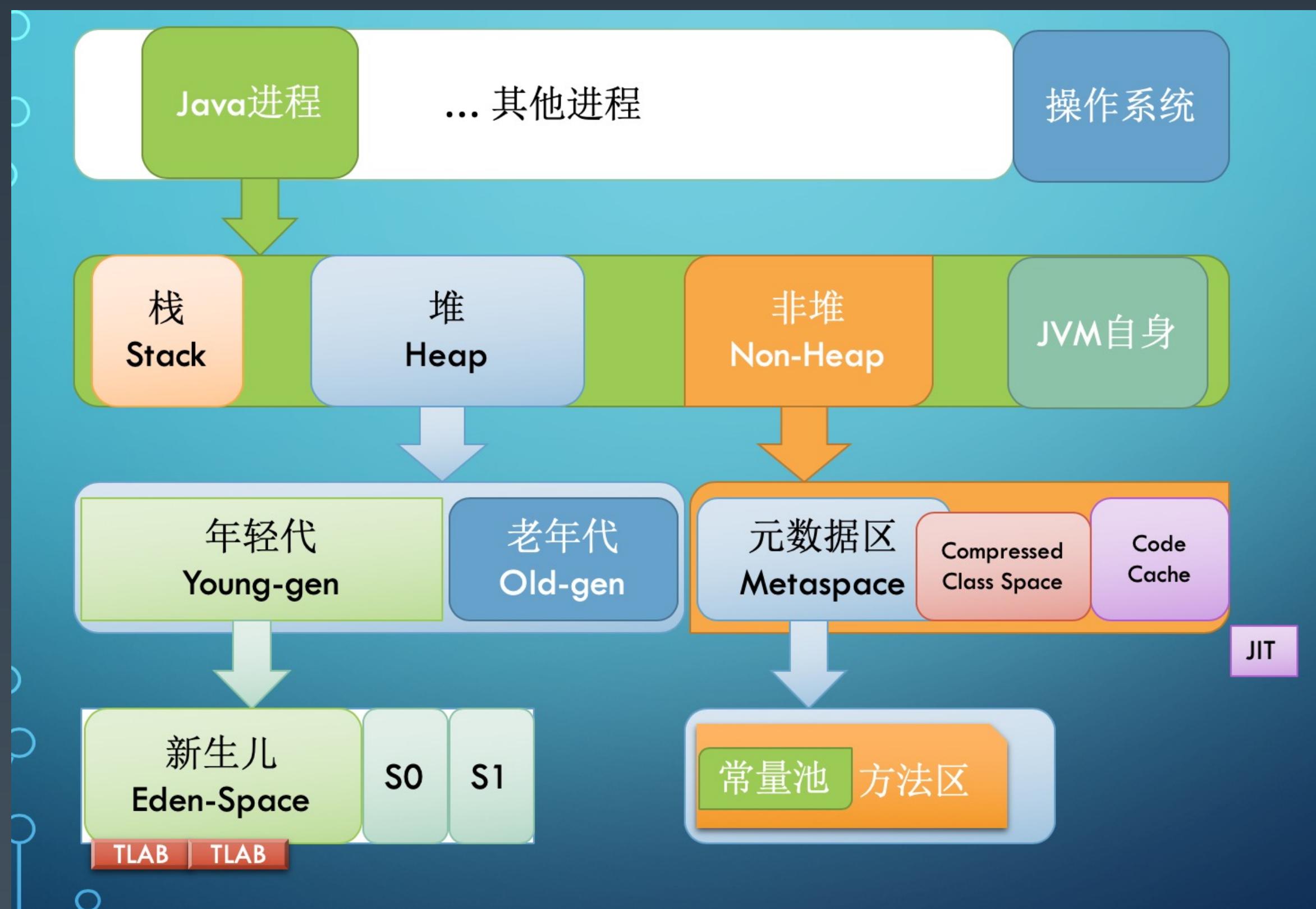
JVM 栈内存结构



栈帧是一个逻辑上的概念，具体的大小在一个方法编写完成后基本上就能确定。

比如返回值需要有一个空间存放，每个局部变量都需要对应的地址空间，此外还有给指令使用的操作数栈，以及 class 指针（标识这个栈帧对应的是哪个类的方法，指向非堆里面的 Class 对象）。

JVM 堆内存结构



堆内存是所有线程共用的内存空间，JVM 将 Heap 内存分为年轻代（Young generation）和 老年代（Old generation, 也叫 Tenured）两部分。

年轻代还划分为 3 个内存池，新生代（Eden space）和存活区（Survivor space），在大部分 GC 算法中有 2 个存活区（S0, S1），在我们可以在观察到的任何时刻，S0 和 S1 总有一个是空的，但一般较小，也不浪费多少空间。

Non-Heap 本质上还是 Heap，只是一般不归 GC 管理，里面划分为 3 个内存池。

Metaspace，以前叫持久代（永久代，Permanent generation），Java8 换了个名字叫 Metaspace。

CCS, Compressed Class Space, 存放 class 信息的，和 Metaspace 有交叉。

Code Cache 存放 JIT 编译器编译后的本地机器代码。

小结：什么是 JMM？

JMM 规范对应的是 “[JSR-133. Java Memory Model and Thread Specification]” , 《Java 语言规范》的 [§17.4. Memory Model 章节]

JMM 规范明确定义了不同的线程之间，通过哪些方式，在什么时候可以看见其他线程保存到共享变量中的值；以及在必要时，如何对共享变量的访问进行同步。这样的好处是屏蔽各种硬件平台和操作系统之间的内存访问差异，实现了 Java 并发程序真正的跨平台。

所有的对象（包括内部的实例成员变量），static 变量，以及数组，都必须存放到堆内存中。

局部变量，方法的形参/入参，异常处理语句的入参不允许在线程之间共享，所以不受内存模型的影响。

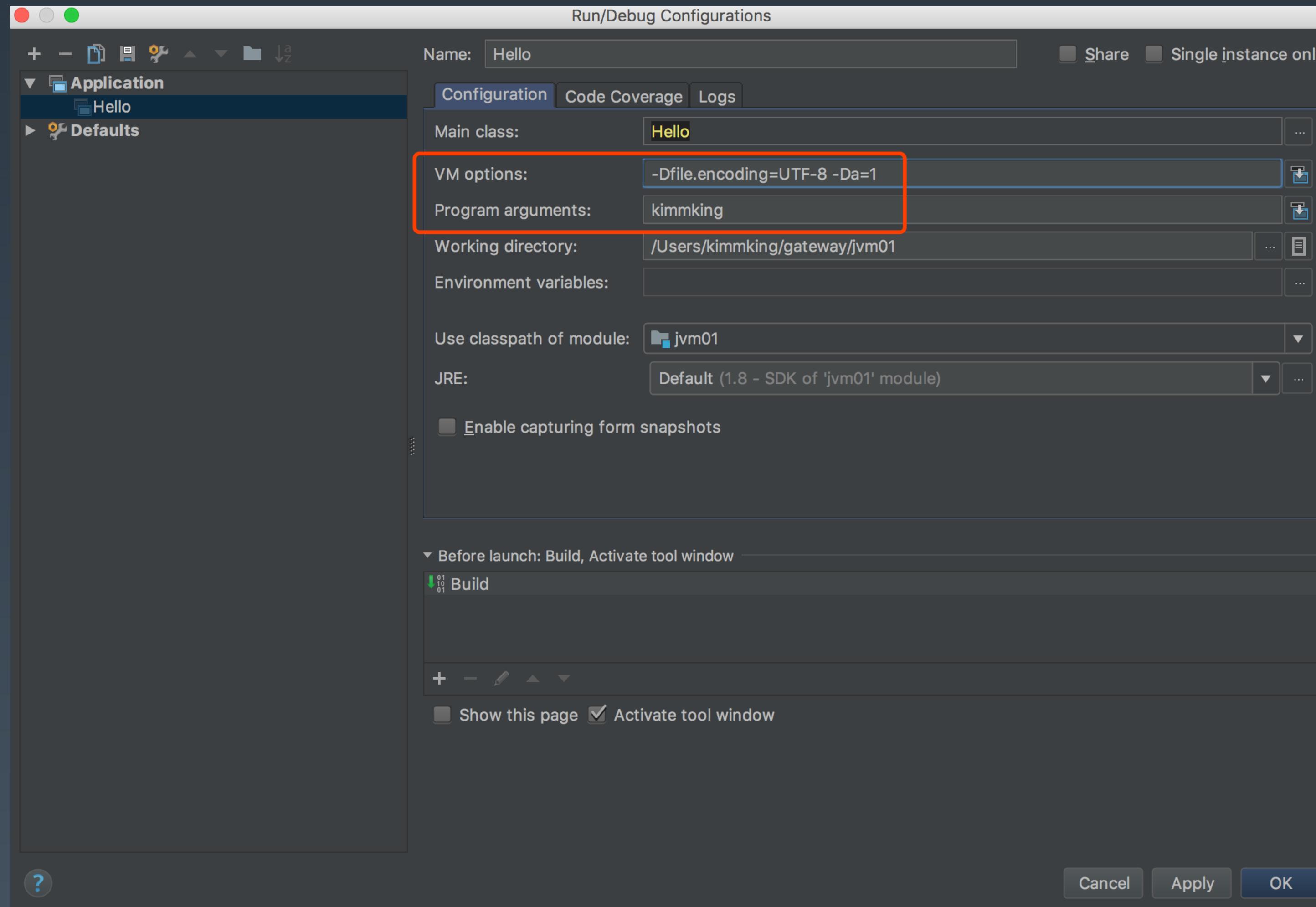
多个线程同时对一个变量访问时【读取/写入】，这时候只要有某个线程执行的是写操作，那么这种现象就称之为“冲突”。

可以被其他线程影响或感知的操作，称为线程间的交互行为，可分为：读取、写入、同步操作、外部操作等等。其中同步操作包括：对 volatile 变量的读写，对管程（monitor）的锁定与解锁，线程的起始操作与结尾操作，线程启动和结束等等。外部操作则是指对线程执行环境之外的操作，比如停止其他线程等等。

JMM 规范的是线程间的交互操作，而不管线程内部对局部变量进行的操作。

5. JVM 启动参数

JVM 启动参数



java [options] classname [args]
java [options]-jar filename [args]

```
D:\git>jps -v  
25196 Jps -Dapplication.home=D:\tools\jdk8 -Xms8m  
  
D:\git>jps -m  
21132 Jps -m
```

JVM 启动参数

```
-server  
-Dfile.encoding=UTF-8  
    -Xmx8g  
-XX:+UseG1GC  
-XX:MaxPermSize=256m
```

以 `-` 开头为标准参数，所有的 JVM 都要实现这些参数，并且向后兼容。

`-D` 设置系统属性。

以 `-X` 开头为非标准参数，基本都是传给 JVM 的，默认 JVM 实现这些参数的功能，但是并不保证所有 JVM 实现都满足，且不保证向后兼容。可以使用 `java -X` 命令来查看当前 JVM 支持的非标准参数。

以 `-XX:` 开头为非稳定参数，专门用于控制 JVM 的行为，跟具体的 JVM 实现有关，随时可能会在下个版本取消。

`-XX: +-Flags` 形式，`+-` 是对布尔值进行开关。

`-XX: key=value` 形式，指定某个选项的值。

JVM 启动参数

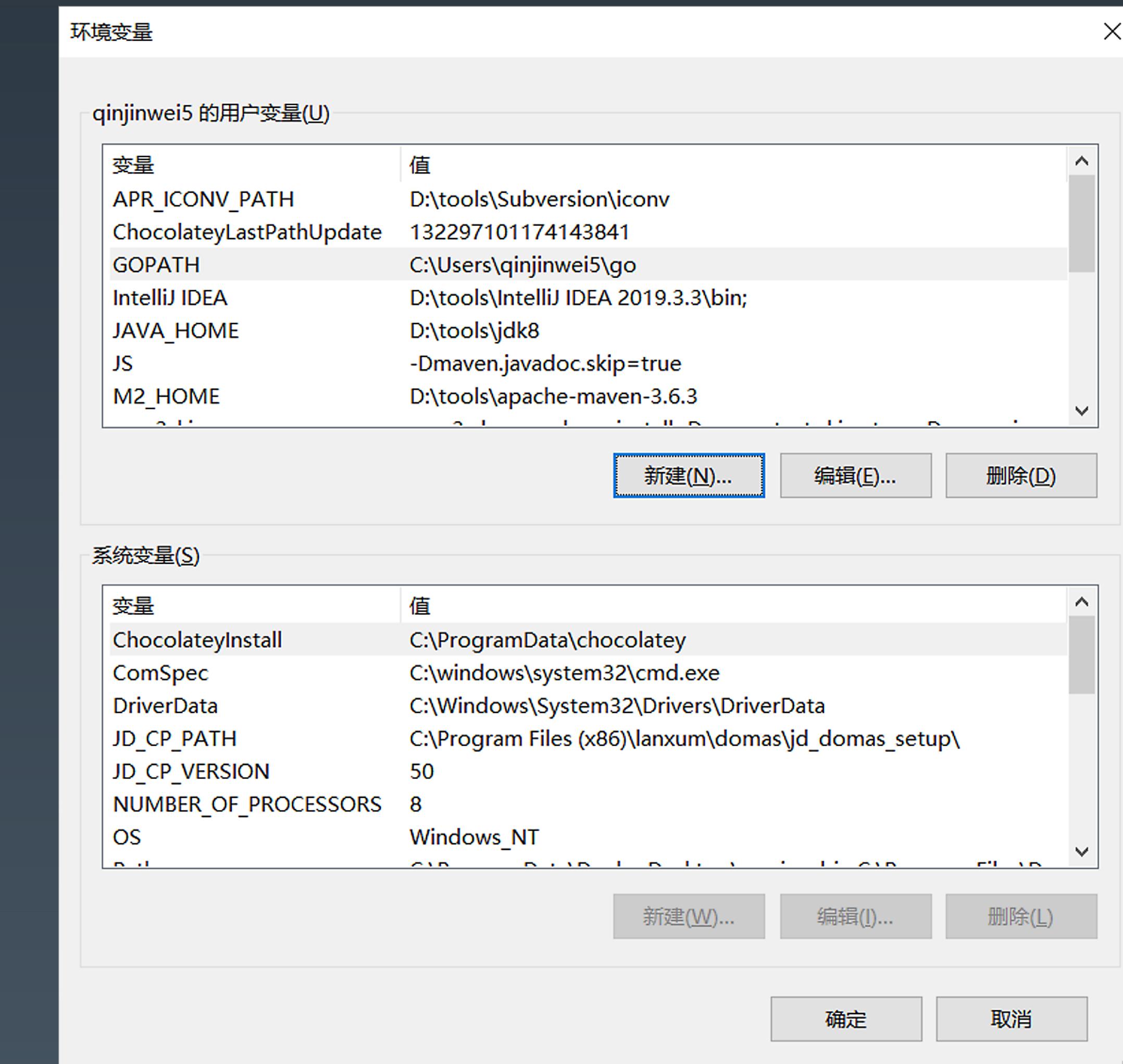
1. 系统属性参数
2. 运行模式参数
3. 堆内存设置参数
4. GC 设置参数
5. 分析诊断参数
6. JavaAgent 参数

JVM 启动参数--系统属性

```
-Dfile.encoding=UTF-8  
-Duser.timezone=GMT+08  
-Dmaven.test.skip=true  
-Dio.netty.eventLoopThreads=8
```

```
System.setProperty("a","A100");  
String a=System.getProperty("a");
```

Linux 上还可以通过: a=A100 java XXX



JVM 启动参数--运行模式

1. **-server**: 设置 JVM 使用 server 模式，特点是启动速度比较慢，但运行时性能和内存管理效率很高，适用于生产环境。在具有 64 位能力的 JDK 环境下将默认启用该模式，而忽略 -client 参数。
2. **-client** : JDK1.7 之前在32位的 x86 机器上的默认值是 -client 选项。设置 JVM 使用 client 模式，特点是启动速度比较快，但运行时性能和内存管理效率不高，通常用于客户端应用程序或者 PC 应用开发和调试。此外，我们知道 JVM 加载字节码后，可以解释执行，也可以编译成本地代码再执行，所以可以配置 JVM 对字节码的处理模式。
3. **-Xint**: 在解释模式 (interpreted mode) 下运行，-Xint 标记会强制 JVM 解释执行所有的字节码，这当然会降低运行速度，通常低10倍或更多。
4. **-Xcomp**: -Xcomp 参数与-Xint 正好相反，JVM 在第一次使用时会把所有的字节码编译成本地代码，从而带来最大程度的优化。【注意预热】
5. **-Xmixed**: -Xmixed 是混合模式，将解释模式和编译模式进行混合使用，有 JVM 自己决定，这是 JVM 的默认模式，也是推荐模式。我们使用 `java -version` 可以看到 mixed mode 等信息。

JVM 启动参数--堆内存

堆内 (Xms-Xmx)

非堆+堆外

1. 如果什么都不配置会如何?
2. Xmx 是否与 Xms 设置相等?
3. Xmx 设置为机器内存的什么比例合适?
4. 作业：画一下 Xmx、Xms、Xmn、Meta、
DirectMemory、Xss 这些内存参数的关系

-Xmx, 指定最大堆内存。如 -Xmx4g。这只是限制了 Heap 部分的最大值为 4g。这个内存不包括栈内存，也不包括堆外使用的内存。

-Xms, 指定堆内存空间的初始大小。如 -Xms4g。而且指定的内存大小，并不是操作系统实际分配的初始值，而是GC先规划好，用到才分配。专用服务器上需要保持 -Xms 和 -Xmx 一致，否则应用刚启动可能就有好几个 FullGC。当两者配置不一致时，堆内存扩容可能会导致性能抖动。

-Xmn, 等价于 -XX:NewSize，使用 G1 垃圾收集器 不应该 设置该选项，在其他的某些业务场景下可以设置。官方建议设置为 -Xmx 的 1/2 ~ 1/4。

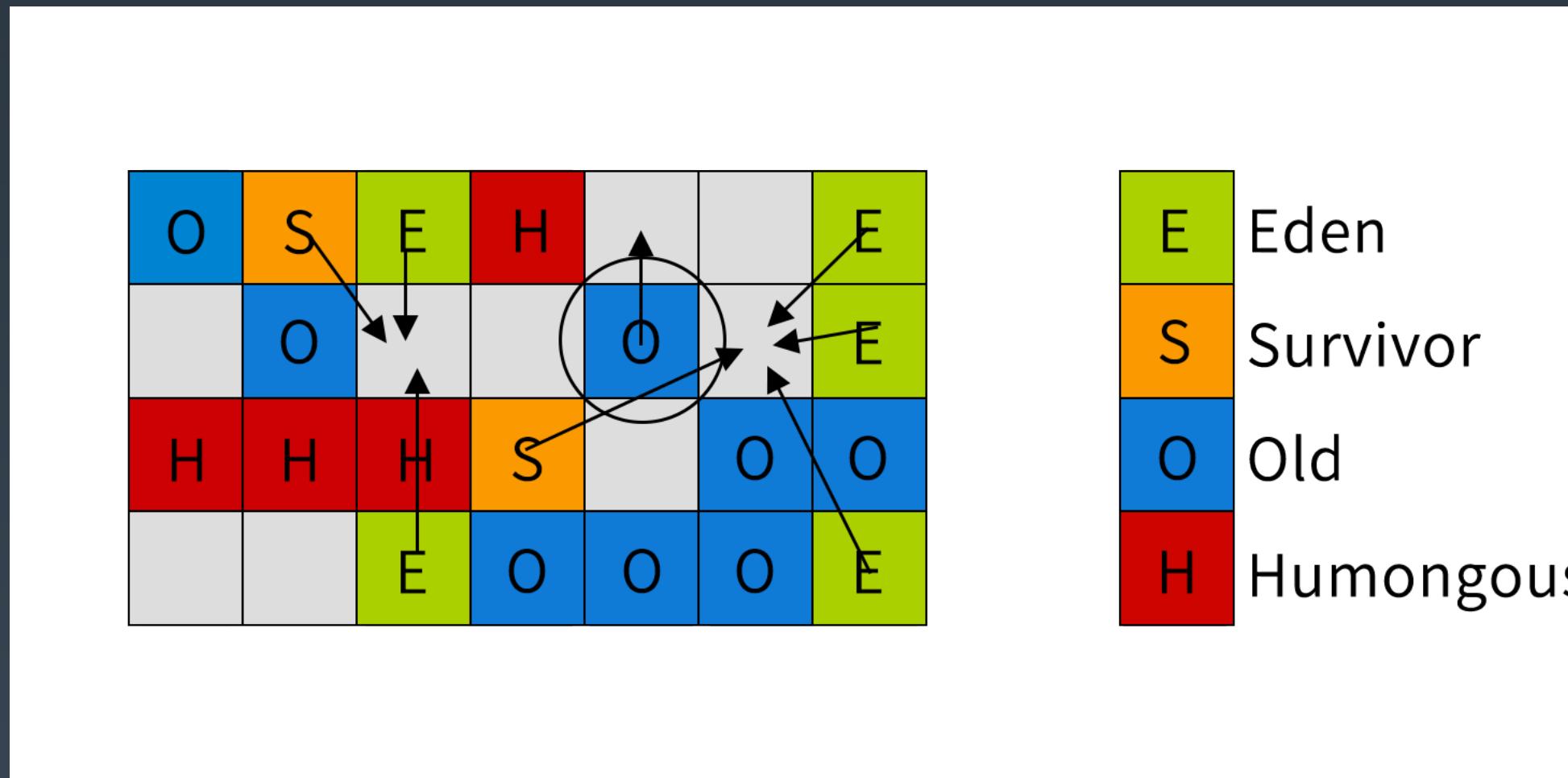
-XX: MaxPermSize=size, 这是 JDK1.7 之前使用的。Java8 默认允许的 Meta 空间无限大，此参数无效。

-XX: MaxMetaspaceSize=size, Java8 默认不限制 Meta 空间，一般不允许设置该选项。

-XX: MaxDirectMemorySize=size, 系统可以使用的最大堆外内存，这个参数跟 -Dsun.nio.MaxDirectMemorySize 效果相同。

-Xss, 设置每个线程栈的字节数，影响栈的深度。例如 -Xss1m 指定线程栈为 1MB，与 -XX:ThreadStackSize=1m 等价。

JVM 启动参数 -- GC 相关



-XX: +UseG1GC: 使用 G1 垃圾回收器

-XX: +UseConcMarkSweepGC: 使用 CMS 垃圾回收器

-XX: +UseSerialGC: 使用串行垃圾回收器

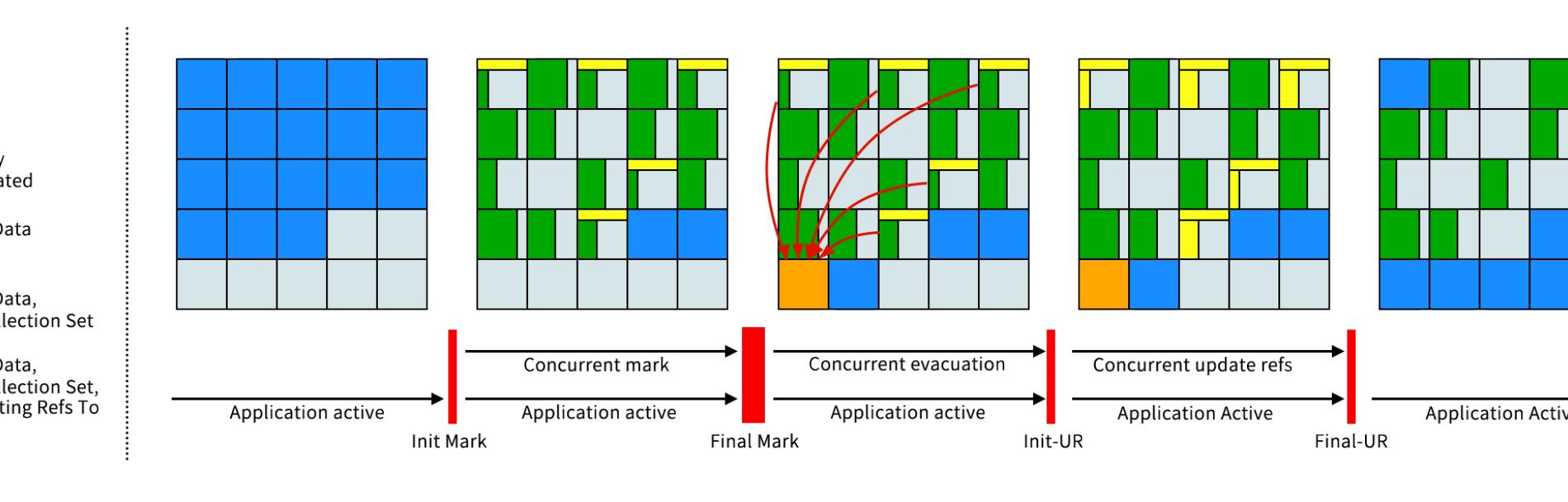
-XX: +UseParallelGC: 使用并行垃圾回收器

// Java 11+

-XX: +UnlockExperimentalVMOptions -XX:+UseZGC

// Java 12+

-XX: +UnlockExperimentalVMOptions -XX:+UseShenandoahGC



1. 各个 JVM 版本的默认 GC 是什么？

JVM 启动参数--分析诊断

-XX: +HeapDumpOnOutOfMemoryError 选项，当 OutOfMemoryError 产生，即内存溢出（堆内存或持久代）时，自动 Dump 堆内存。

示例用法： `java -XX:+HeapDumpOnOutOfMemoryError -Xmx256m ConsumeHeap`

-XX: HeapDumpPath 选项，与 HeapDumpOnOutOfMemoryError 搭配使用，指定内存溢出时 Dump 文件的目录。

如果没有指定则默认为启动 Java 程序的工作目录。

示例用法： `java -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/local/ ConsumeHeap`

自动 Dump 的 hprof 文件会存储到 /usr/local/ 目录下。

-XX: OnError 选项，发生致命错误时（fatal error）执行的脚本。

例如，写一个脚本来记录出错时间，执行一些命令，或者 curl 一下某个在线报警的 url。

示例用法： `java -XX:OnError="gdb - %p" MyApp`

可以发现有一个 %p 的格式化字符串，表示进程 PID。

-XX: OnOutOfMemoryError 选项，抛出 OutOfMemoryError 错误时执行的脚本。

-XX: ErrorFile=filename 选项，致命错误的日志文件名，绝对路径或者相对路径。

`-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=1506`，远程调试。

JVM 启动参数 ---JavaAgent

Agent 是 JVM 中的一项黑科技，可以通过无侵入方式来做很多事情，比如注入 AOP 代码，执行统计等等，权限非常大。这里简单介绍一下配置选项，详细功能需要专门来讲。

设置 agent 的语法如下：

`-agentlib:libname[=options]` 启用 native 方式的 agent，参考 LD_LIBRARY_PATH 路径。

`-agentpath:pathname[=options]` 启用 native 方式的 agent。

`-javaagent:jarpath[=options]` 启用外部的 agent 库，比如 pinpoint.jar 等等。

`-Xnoagent` 则是禁用所有 agent。

以下示例开启 CPU 使用时间抽样分析：

```
JAVA_OPTS="-agentlib:hprof(cpu=samples,file=cpu.samples.log)"
```

6. 总结回顾与作业实践

第1课总结回顾

字节码技术

类加载器

内存模型

启动参数

第1课作业实践

1. (可选) 自己写一个简单的 Hello.java，里面需要涉及基本类型、四则运算、if 和 for，然后自己分析一下对应的字节码，有问题群里讨论。
2. (必做) 自定义一个 Classloader，加载一个 Hello.xlass 文件，执行 hello 方法，此文件内容是一个 Hello.class 文件所有字节 ($x=255-x$) 处理后的文件。文件在视频下方下载。
3. (必做) 画一张图，展示 Xmx、Xms、Xmn、Metaspache、DirectMemory、Xss 这些内存参数的关系。
4. (可选) 检查一下自己维护的业务系统的 JVM 参数配置，用 jstat 和 jstack、jmap 查看一下详情，并且自己独立分析一下大概情况，思考有没有不合理的地方，如何改进。

注意：

1. 对于线上有流量的系统，慎重使用 jmap 命令。
2. 如果没有线上系统，可以自己 run 一个 web/java 项目。或者直接查看 idea 进程。

课堂重点知识笔记和答疑链接，请在群里找或者向班主任索要。

THANKS! | 极客大学