03-JVM虚拟机

刘亚雄

极客时间-Java 讲师

今日目标

1. 理解JVM是什么
2. 深入理解Java内存区域，也就是运行时数据区
3. 掌握内存划分的基本理论依据：三大假说
4. 理解为什么要划分年轻代和老年代

JVM虚拟机的灵魂三问

JVM是什么？

- 广义上指的是一种规范，狭义上的是JDK中的JVM虚拟机（ Hotspot 、zing、j9等等）。

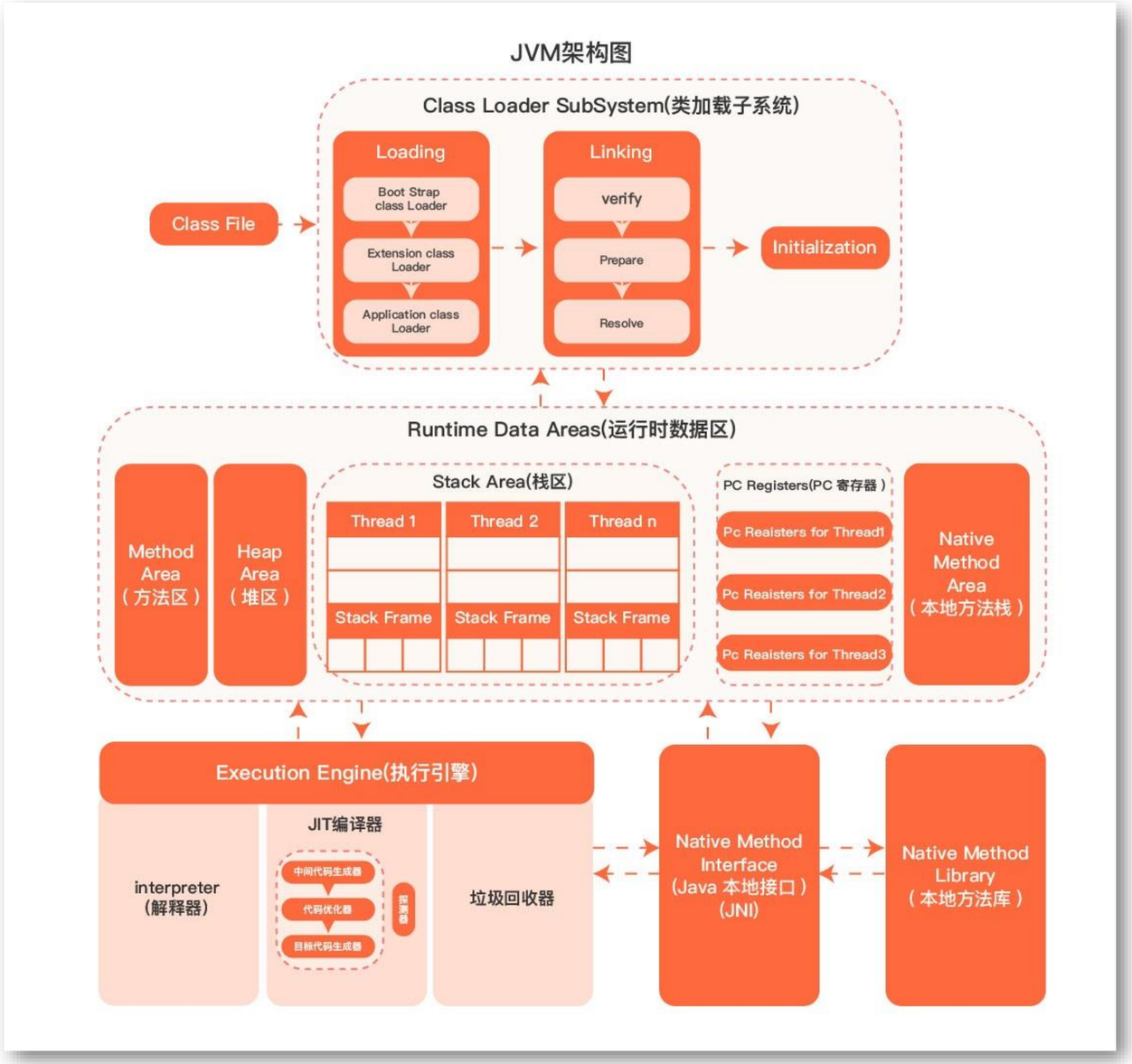
为什么要学习JVM？

- 面试高频考点
- 研发中重难点问题与总会与JVM有关系。例如：线程死锁、内存溢出、项目性能优化等等。
- 基础不牢，地动山摇：想深入掌握Java这么语言，JVM始终绕不过去的那座大山，早晚得攀登。

怎么学习JVM？

- **JVM基本常识 → 类加载系统 → 运行时数据区 → 一个对象的一生 → GC收集器 →实战**

JVM架构图


JVM架构图
Class Loader SubSystem(类加载子系统)
Loading
Boot Strap class Loader
Extension class Loader
Application class Loader
Linking
verify
Prepare
Resolve
Class File
Initialization
Runtime Data Areas(运行时数据区)
Method Area (方法区)
Heap Area (堆区)
Stack Area(栈区)
Thread 1
Thread 2
Thread n
Stack Frame
Stack Frame
Stack Frame
PC Registers(PC 寄存器)
Pc Reaisters for Thread1
Pc Reaisters for Thread2
Pc Reaisters for Thread3
Native Method Area (本地方法栈)
Execution Engine(执行引擎)
interpreter (解释器)
JIT编译器
中间代码生成器
代码优化器
目标代码生成器
探查器
垃圾回收器
Native Method Interface (Java 本地接口) (JNI)
Native Method Library (本地方法库)

一、JVM虚拟机基本常识

1.1 什么是JVM?

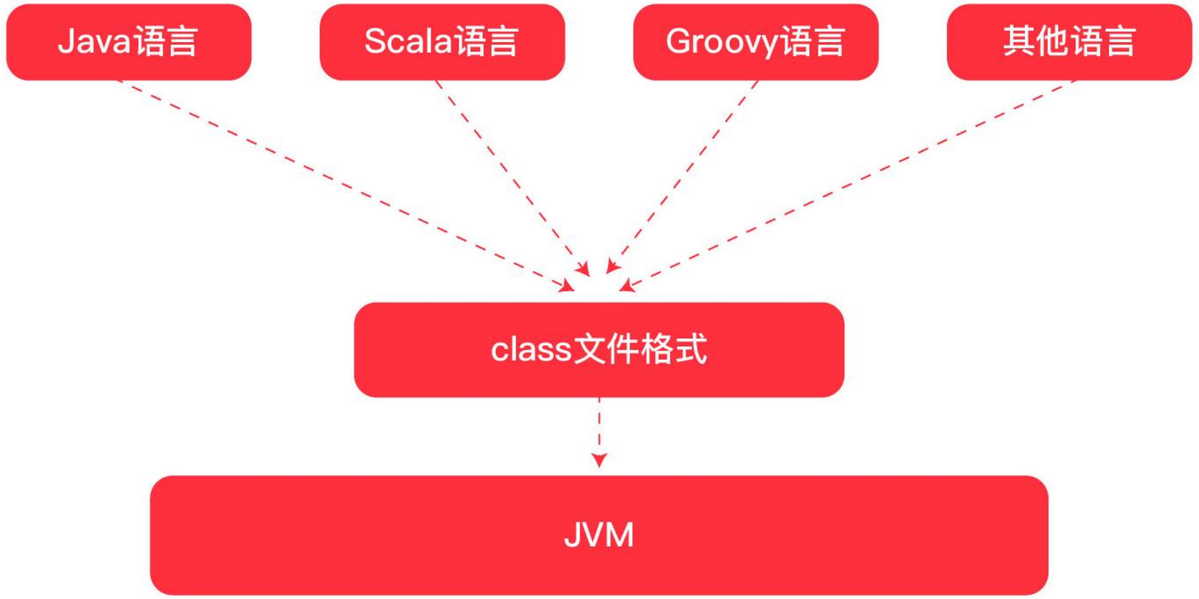
平时我们所说的JVM广义上指的是一种规范。狭义上的是JDK中的JVM虚拟机。JVM的实现是由各个厂商来做的。比如：hotspot属于Oracle公司、JRocket、IBM j9、zing 、taobao.vm等等。可以这么说Java，Kotlin、Clojure、JRuby、Groovy等运行于Java虚拟机上的编程语言及其相关的程序都属于Java技术体系中的一员。

01-Java技术体系：

- Java语言
- Java类库API
- 第三方类库：Google、Apache、其他
- **Java虚拟机：各种硬件平台上的Java虚拟机实现**

1.1 什么是JVM？

02-Java和JVM的关系

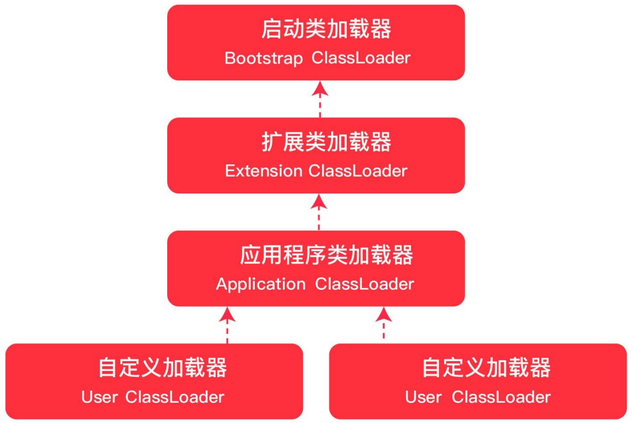


二、类加载系统

2.1 类加载器

JVM的类加载是通过ClassLoader及其子类来完成的：那么有哪些类加载器呢？

- **启动类加载器(Bootstrap ClassLoader)**
  - 负责加载 JAVA_HOME\lib 目录的或通过-Xbootclasspath参数指定路径中的且被虚拟机认可（rt.jar）的类库
- **扩展类加载器(Extension ClassLoader)**
  - 负责加载 JAVA_HOME\lib\ext 目录或通过java.ext.dirs系统变量指定路径中的类库
- **应用程序类加载器(Application ClassLoader)**
  - 负责加载用户路径classpath上的类库
- **自定义类加载器（User ClassLoader）**
  - 加载应用之外的类文件
  - **举个栗子：JRebel**

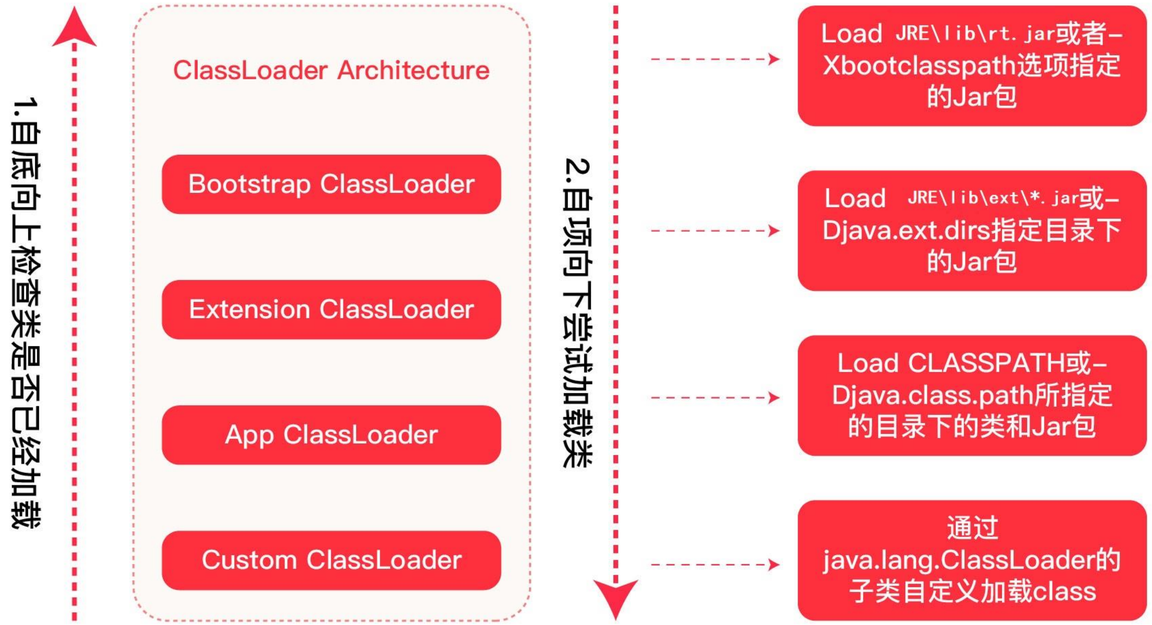


类加载器的执行顺序是什么？

2.2 执行顺序

1.检查顺序是自底向上：加载过程中会先检查类是否被已加载，从Custom到BootStrap逐层检查，只要某个类加载器已加载就视为此类已加载，保证此类所有ClassLoader只加载一次

2.加载的顺序是自顶向下：也就是由上层来逐层尝试加载此类。



什么时候加载类呢？

2.3 加载时机与过程

01-类加载的四时机：

- 01-遇到new、getStatic、putStatic、invokeStatic四条指令时
- 02-使用java.lang.reflect包方法时，对类进行反射调用
- 03-初始化一个类时，发现其父类还没初始化，要先初始化其父类
- 04-当虚拟机启动时，用户需要指定一个主类main，需要先将主类加载

```java
public class Student{
    private static int age ;
    public static void method(){
    }
}

//Student.age
//Student.method();
//new Student();
```

02-一个类的一生：

```java
Class c = Class.forname("com.hero.Student");
```

03-类加载做了什么？主要做三件事：

- 01-类全限定名称 → 二进制字节流加载class文件
- 02-字节流静态数据 → 方法区（永久代，元空间）
- 03-创建字节码Class对象

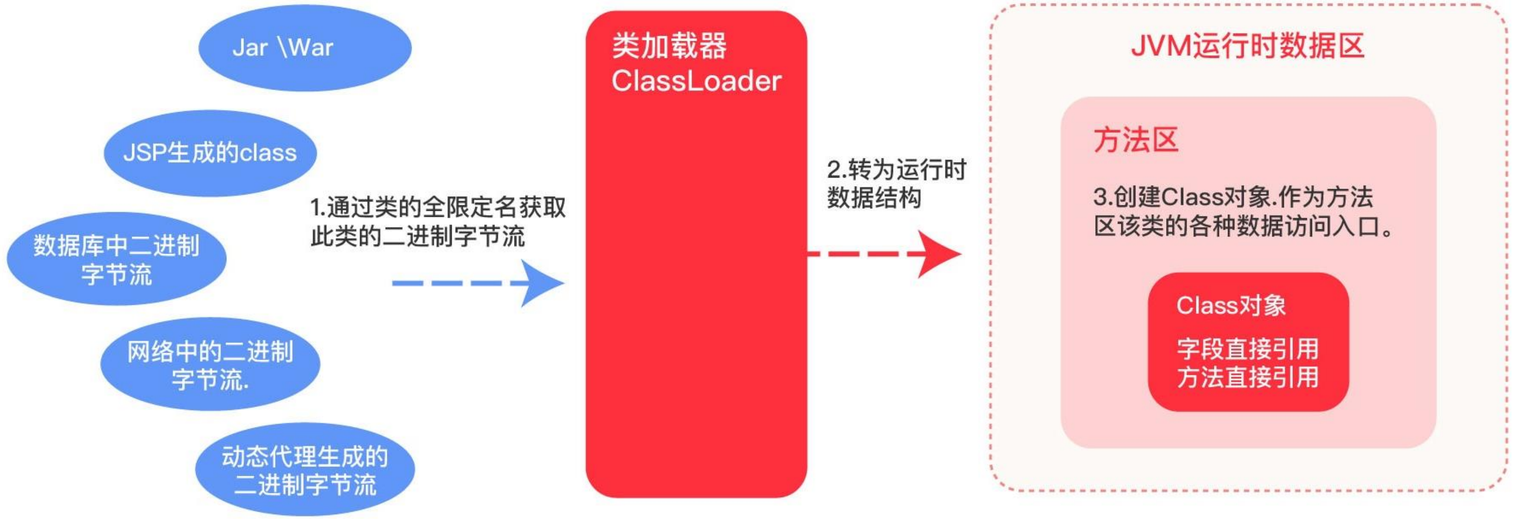
类的生命周期

可以从哪些途径加载类呢？

2.4 类加载途径

加载途径

- 01-jar/war
- 02-jsp生成的class
- 03-数据库中的二进制字节流
- 04-网络中的二进制字节流
- 05-动态代理生成的二进制字节流



手写一个类加载器

2.5 自定义类加载器

目标：自定义类加载器，加载指定路径在D盘下的lib文件夹下的类。

步骤：

1. 新建一个类Test.jave
2. 编译Test.jave到指定lib目录
3. 自定义类加载器HeroClassLoader继承ClassLoader：
   - 重写findClass()方法
   - 调用defineClass()方法
4. 测试自定义类加载器

2.6 双亲委派模型与打破双亲委派

01-什么是双亲委派？

- **当一个类加载器收到类加载任务，会先交给其父类加载器去完成**，因此最终加载任务都会传递到顶层的启动类加载器，只有当父类加载器无法完成加载任务时，才会尝试执行加载任务

02-为什么需要双亲委派呢？

- 主要考虑安全因素，双亲委派可以避免重复加载核心的类，当父类加载器已经加载了该类时，子类加载器不会再去加载
- 比如：要加载位于rt.jar包中的类java.lang.Object，不管是哪个加载器加载，最终都委托给顶层的启动类加载器进行加载，这样就可以保证使用不同的类加载器最终得到的都是同样的Object对象。

双亲委派是如何实现的呢？

2.6 双亲委派模型与打破双亲委派

03-为什么还需要破坏双亲委派？

- 在实际应用中，**双亲委派解决了Java 基础类统一加载的问题，但是却存在着缺陷**。JDK中的基础类作为典型的API被用户调用，但是也存在**API调用用户代码**的情况，典型的如：SPI代码。这种情况就需要打破双亲委派模式。

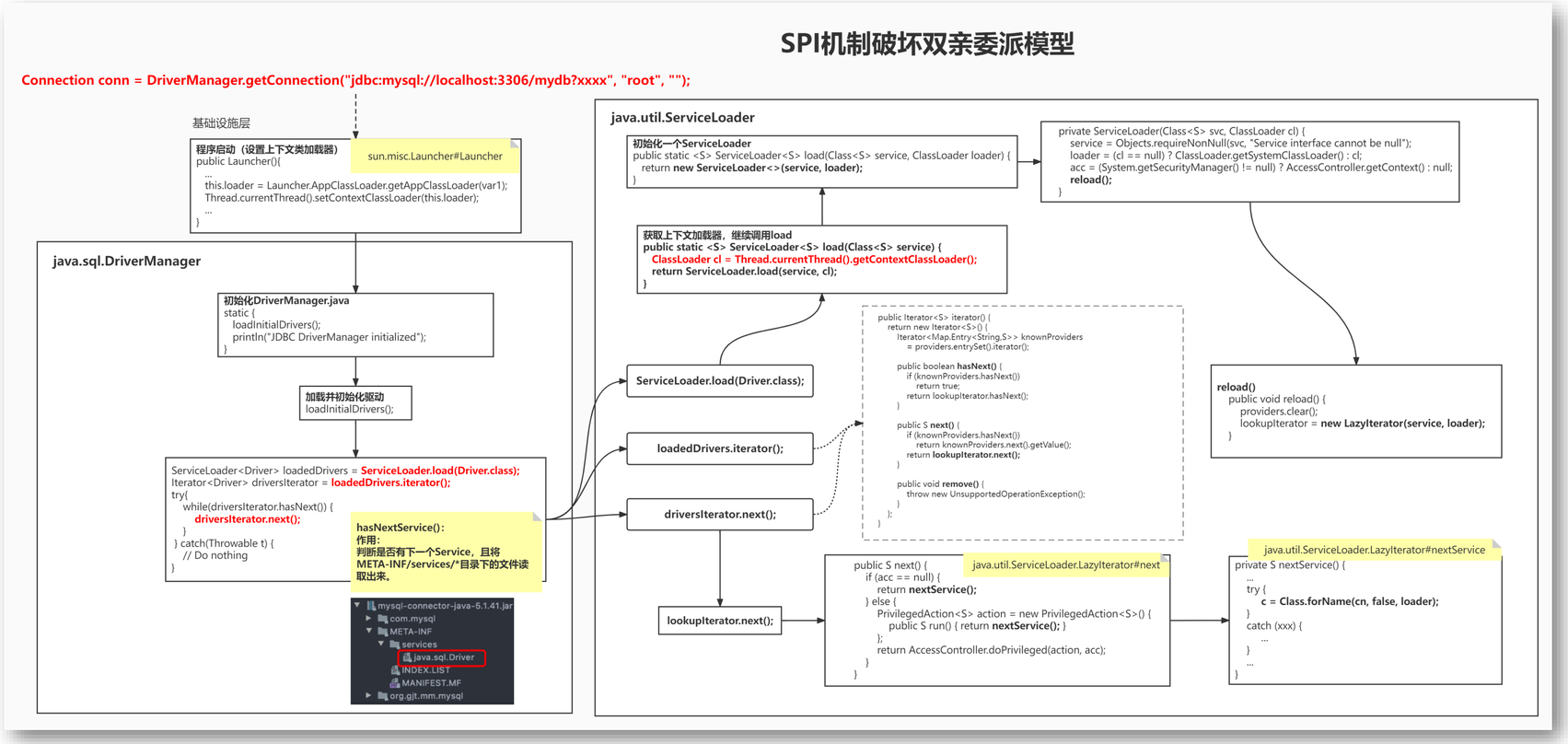
- 数据库驱动DriverManager。以Driver接口为例，Driver接口定义在JDK中，其**实现由各个数据库的服务商来提供，由系统类加载器加载**。这个时候就需要 **启动类加载器** 来 **委托** 子类来加载Driver实现，这就破坏了双亲委派。

还有哪些破坏双亲委派的方式呢？

2.6 双亲委派模型与打破双亲委派

04-如何破坏双亲委派？

- 方式一：重写ClassLoader的loadClass方法
- 方式二：SPI，父类委托自类加载器加载Class，以数据库驱动DriverManager为例
- 方式三：热部署和不停机更新用到的OSGI技术



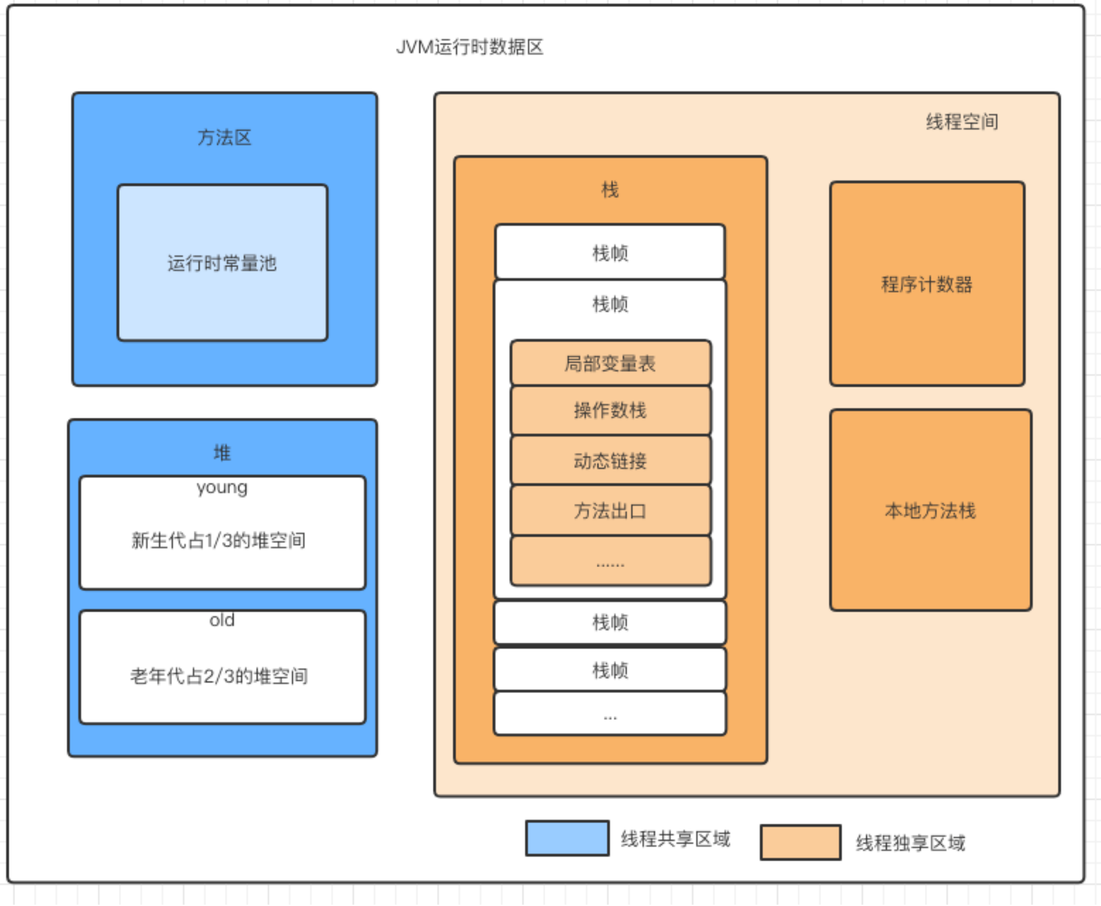
三、运行时数据区

3.1 运行时数据区概述

整个JVM构成里面，主要由三部分组成：类加载系统、**运行时数据区**、执行引擎

按照线程使用情况和职责分成两大类：

- **线程独享 （程序执行区域）**
  - **虚拟机栈**、本地方法栈、**程序计数器**
  - 不需要垃圾回收
- **线程共享 （数据存储区域）**
  - **堆**和**方法区**
  - 存储类的静态数据和对象数据
  - 需要垃圾回收



3.2 运行时数据区

以下内容请查看课堂笔记：

- 堆
- 虚拟机栈
- 本地方法栈
- 方法区
- 字符串常量池
- 程序计数器
- 直接内存

3.2 运行时数据区-字符串常量池

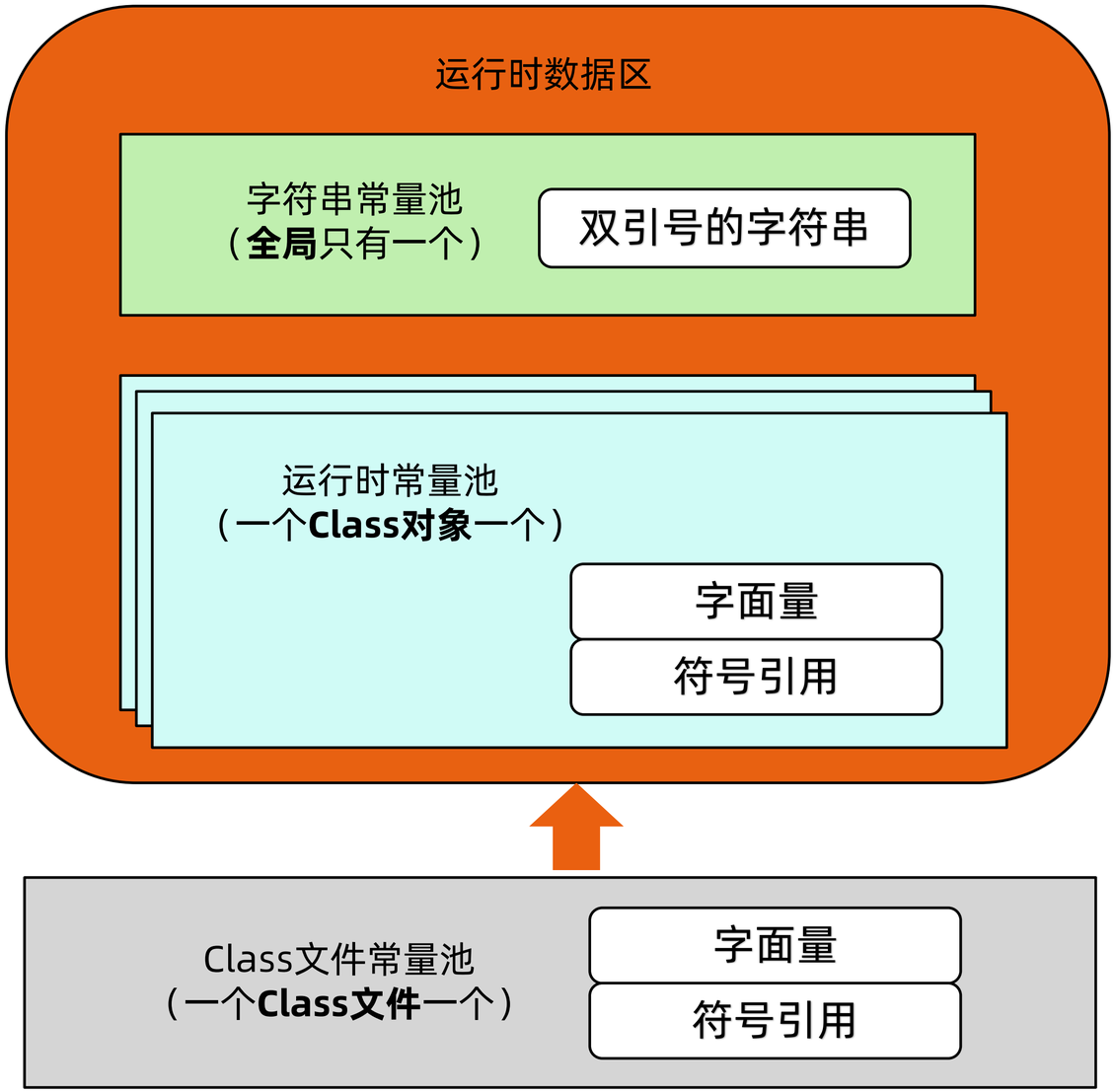
01-三种常量池：

- Class文件常量池
- 运行时常量池
- 字符串常量池

02-字面量与符号引用：

- 字面量：int、float、long、double，**双引号字符串**等
- 符号引用：Class、Method，Field等

字符串常量池如何保存数据？



3.2 运行时数据区-字符串常量池

03-字符串常量池存储数据的方式：

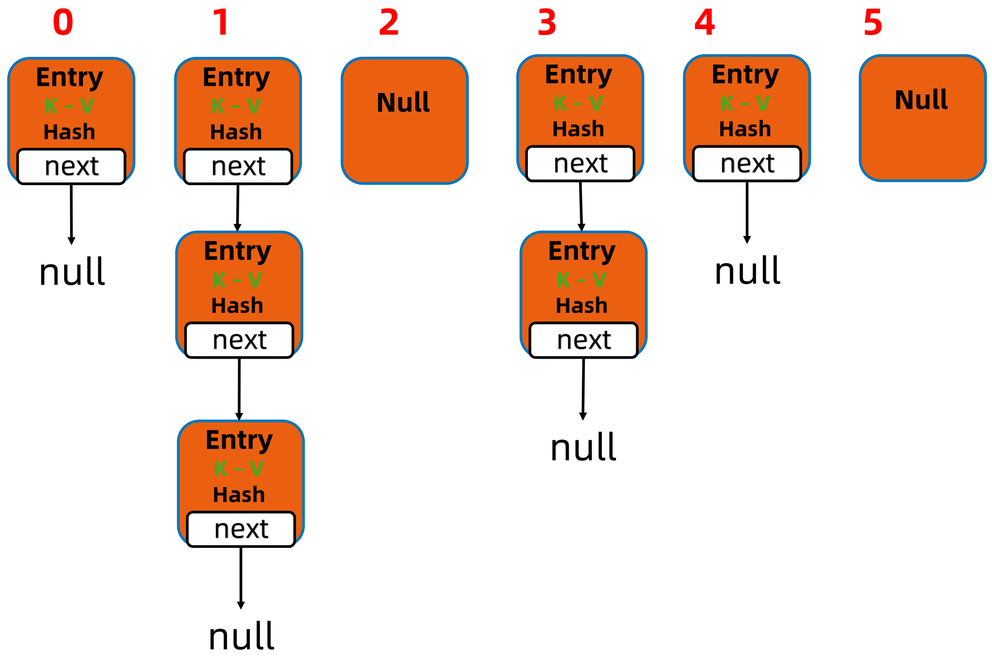
字符串常量池使用的是StringTable的数据结构存储数据，类似于HashTable（哈希表）

04-什么是哈希表呢？

哈希表（也叫散列表），是根据关键码值(K-V)而直接进行访问的数据结构。**本质上就是个数组+链表**

- key：散列函数，**公式：hash(字符串) % 数组size**
- value：字符串的引用
- size：-XX:StringTableSize=65536

目标：加速查找速度



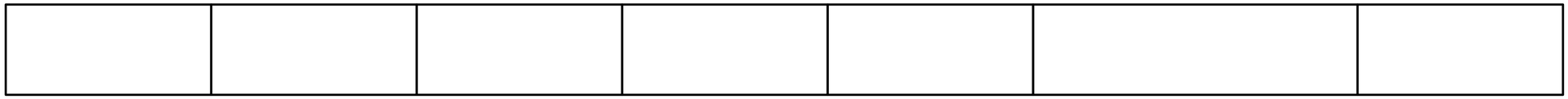
会不会出现散列函数计算索引位置相同的情况？

3.2 运行时数据区-字符串常量池

字符串常量池案例-01

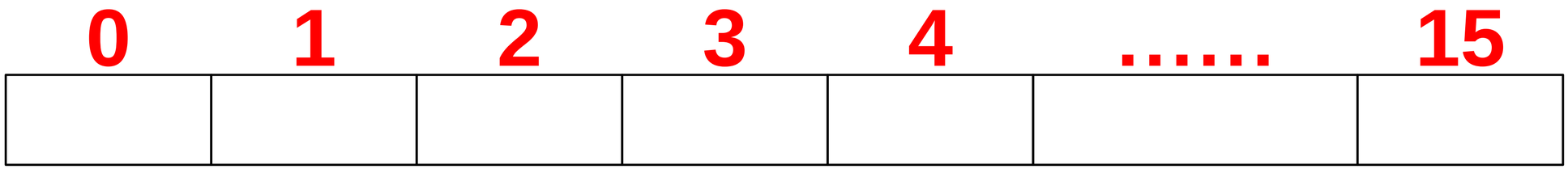
```
public static void main(String[] args) {
    HashMap<String, Integer> map = new HashMap<>();
    map.put("hello", 53);
    map.put("world", 35);
    map.put("java", 55);
    map.put("world", 52);
    map.put("通话", 51);
    map.put("重地", 55);
    System.out.println("hello.hashCode() = " + "hello".hashCode());//99162322
    System.out.println("world.hashCode() = " + "world".hashCode());//113318802
    System.out.println("java.hashCode() = " + "java".hashCode());//3254818
    System.out.println("通话.hashCode() = " + "通话".hashCode());//1179395
    System.out.println("重地.hashCode() = " + "重地".hashCode());//1179395
}
```

字符串常量池案例-01-哈希表存数据



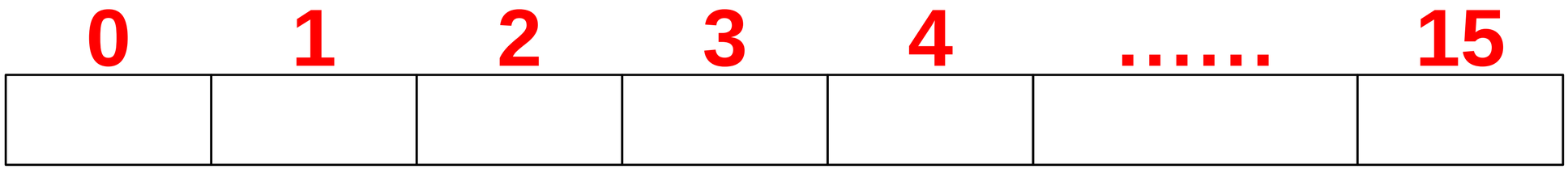
"hello"	99162322
"world"	113318802
"java"	3254818
"world"	113318802
"通话"	1179395
"重地"	1179395

字符串常量池案例-01-哈希表存数据



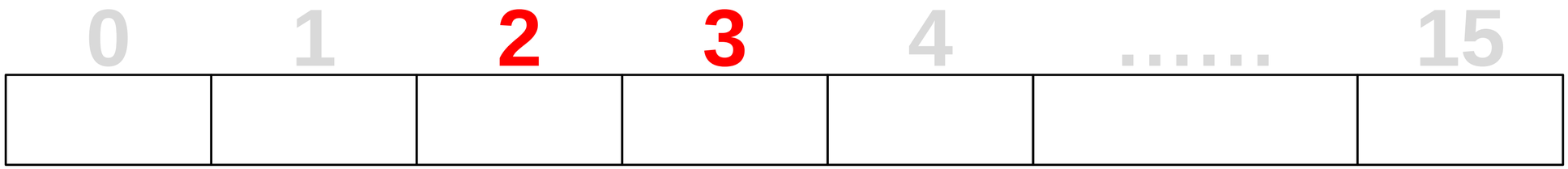
"hello"	99162322
"world"	113318802
"java"	3254818
"world"	113318802
"通话"	1179395
"重地"	1179395

字符串常量池案例-01-哈希表存数据



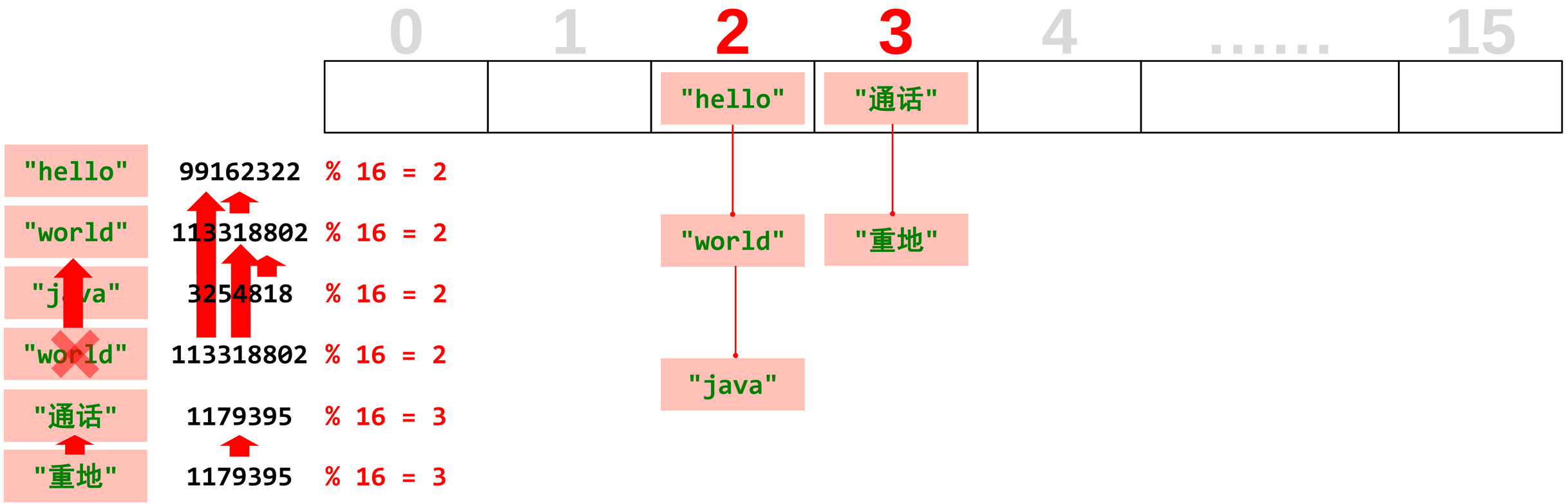
"hello"	99162322	% 16 = 2
"world"	113318802	% 16 = 2
"java"	3254818	% 16 = 2
"world"	113318802	% 16 = 2
"通话"	1179395	% 16 = 3
"重地"	1179395	% 16 = 3

字符串常量池案例-01-哈希表存数据



"hello"	99162322	% 16 = 2
"world"	113318802	% 16 = 2
"java"	3254818	% 16 = 2
"world"	113318802	% 16 = 2
"通话"	1179395	% 16 = 3
"重地"	1179395	% 16 = 3

字符串常量池案例-01-哈希表存数据



3.2 运行时数据区-字符串常量池

字符串常量池案例-02

```
public static void test() {
    String str1 = "abc";
    String str2 = new String("abc");
    System.out.println(str1 == str2);//false

    String str3 = new String("abc");
    System.out.println(str3 == str2);//false

    String str4 = "a" + "b";
    System.out.println(str4 == "ab");//true

    final String s = "a";
    String str5 = s + "b";
    System.out.println(str5 == "ab");//true

    String s1 = "a";
    String s2 = "b";
    String str6 = s1 + s2;
    System.out.println(str6 == "ab");//false

    String str7 = "abc".substring(0,2);
    System.out.println(str7 == "ab");//false

    String str8 = "abc".toUpperCase();
    System.out.println(str8 == "ABC");//false

    String s5 = "a";
    String s6 = "abc";
    String s7 = s5 + "bc";
    System.out.println(s6 == s7.intern());//true
}
```

结论：

- 单独使用**引号(" ")**创建字符串都是常量，编译期存入StringPool
- 使用new创建的字符串对象会存入heap，运行期创建
- 只包含常量的字符串连接符+，编译期存入StringPool
- 含变量的字符串连接符+，运行期创建，存储到Heap
- 运行期调用String的intern()方法，可以动态向StringPool加字符串

3.2 运行时数据区-程序计数器

程序计数器，也叫PC寄存器，当前线程所执行的**字节码指令**的**行号指示器**

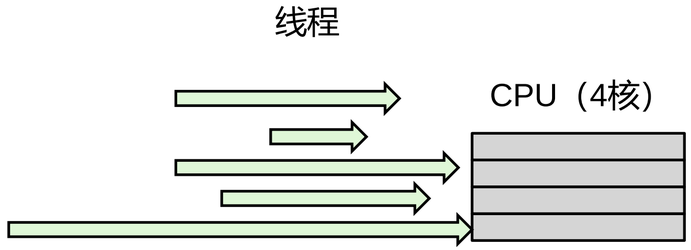
01-为什么需要程序计数器？

- 线程切换（系统上下文切换）后准确恢复执行位置

02-存什么数据？

- Java方法：记录虚拟机字节码指令地址
- Native方法：记录为空

异常：唯一没有OOM异常的区域



3.2 运行时数据区-直接内存

直接内存不是虚拟机运行时数据区的一部分，也不是《Java虚拟机规范》中定义的内存区域

在JDK1.4中，新加入了NIO，引入了Channel和Buffer的IO方式，可以使用native方法直接分片对外内存，然后通过**DirectByteBuffer**对象可以操作直接内存

01-为什么需要直接内存？

因为性能真的好，一会案例验证一下

02-直接内存和堆内存比较：

内存区域	分配空间	读写操作
堆内存	**性能很好**	效率低
直接内存	性能很差	**效率高**

THANKS

教育不是注满一桶水，而是点燃一把火