- **JVM**
  - JVM概述
    - JVM特点
    - JVM的整体结构
    - 总体框架图
    - JVM的架构模型
      - 基于栈式架构
      - 基于寄存器架构
    - 发展历程
      - Sun Classic VM
      - Sun Hotspot VM
  - 类加载器子系统（Class Loader）
    - 作用
    - 类加载过程
      - 1.加载
      - 2.链接
      - 3.初始化
    - 类加载器的分类
      - 引导类加载器（Bootstrap ClassLoader）
      - 自定义类加载器
        - 为什么要自定义类加载器?
    - 获取ClassLoader的途径
    - 双亲委派机制
  - 运行时数据区
    - 程序计数器（PC寄存器）
      - 使用PC寄存器存储字节码指令地址的作用?
      - PC寄存器为什么会被设定为线程私有?
    - 虚拟机栈
      - Java虚拟机栈
        - 特点
        - 问题
      - 栈的存储单位
      - 栈的执行原理
      - 栈帧的内部结构
        - 局部变量表
        - 操作数栈
        - 动态链接
        - 方法返回地址
        - 一些附加信息
      - 相关面试题
    - 本地方法（Native Method）
    - 本地方法栈（Native Method Stack）
  - 堆（Heap）
    - 堆的概述
    - 堆的内存结构
    - 堆空间大小的设置
      - 年轻代&老年代

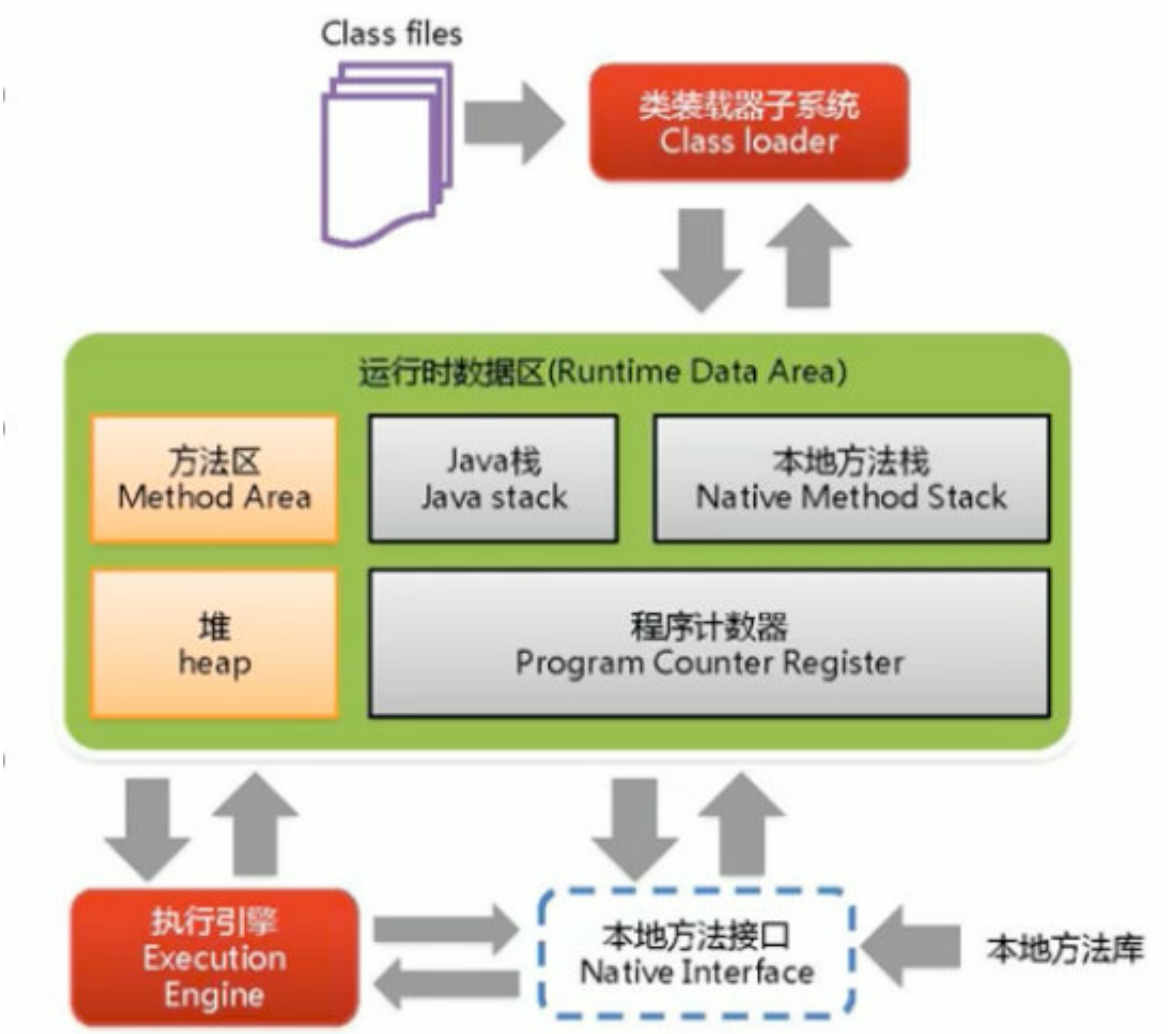
JVM

JVM概述

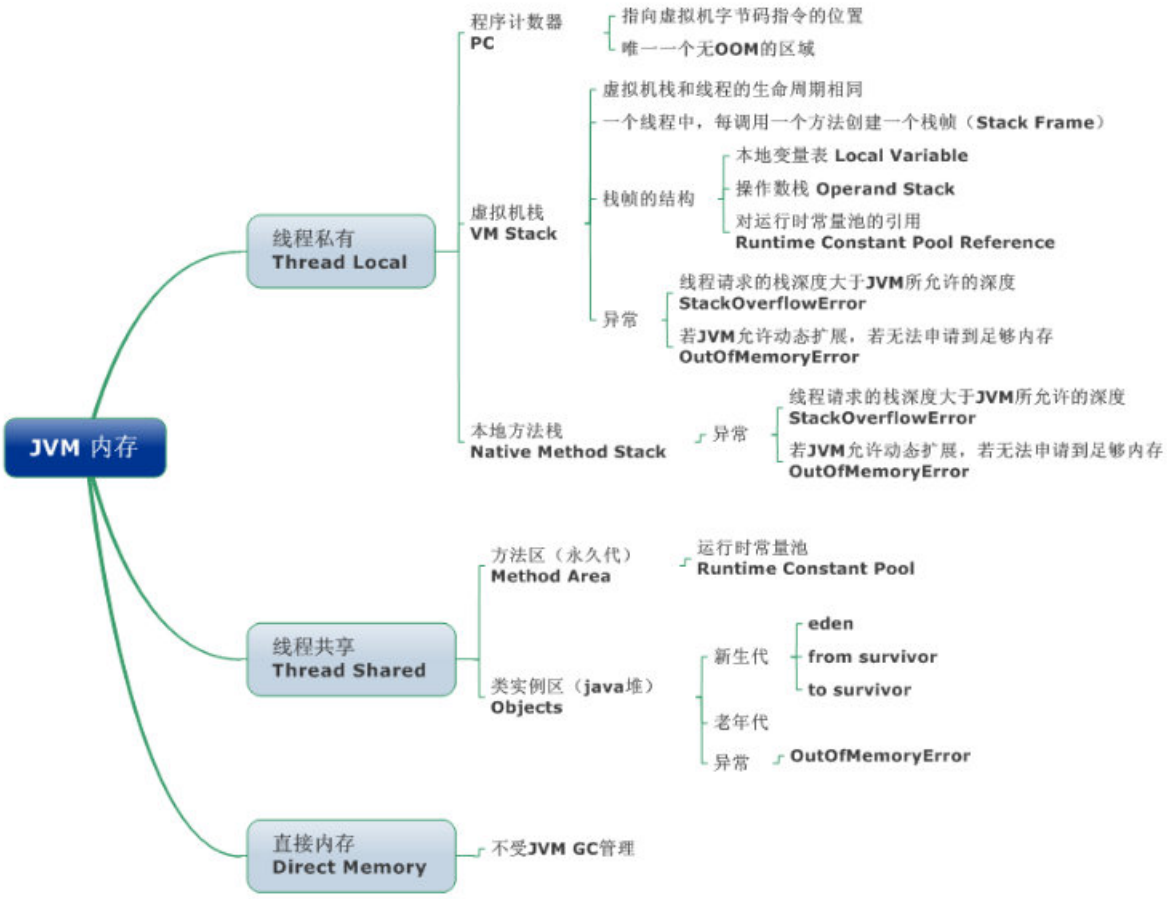
JVM特点

- 一次编译，多处运行
- 自动内存管理
- 自动垃圾回收功能

JVM的整体结构


Class files
类装载器子系统
Class loader
运行时数据区(Runtime Data Area)
方法区
Method Area
Java栈
Java stack
本地方法栈
Native Method Stack
堆
heap
程序计数器
Program Counter Register
执行引擎
Execution
Engine
本地方法接口
Native Interface
本地方法库

总体框架图



JVM的架构模型

基于栈式架构

设计和实现简单，适用于资源受限的系统。不需要硬件支持，可移植性更好。

由于跨平台性，Java的指令都是根据栈设计的

特点：跨平台性，指令集小，指令多，执行性能比寄存器差

基于寄存器架构

完全依赖于硬件，可移植性差

性能优秀，执行更高效

发展历程

Sun Classic VM

世界上第一款商用Java VM

只提供解释器（不包含后端编译器JIT，可以寻找热点代码，存入缓存，提高效率）

现在Hotspot内置了此VM

Sun Hotspot VM

Sun JDK，Open JDK默认VM

通过PC寄存器（程序计数器）找到最具有编译价值的代码，触发即时编译

通过编译器和解释器协同工作，在优化时间与执行性能上取得平衡

类加载器子系统（Class Loader）

作用

1. 负责从文件系统或网络中加载class文件，class文件在文件开头有特殊的文件标识**CAFE BABE**
2. ClassLoader只负责class文件的加载，至于它是否可以运行，由执行引擎决定

类加载过程

1.加载

在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

2.链接

- 验证：确保class文件的字节流中包含的信息符合当前VM要求
- 准备：
  - 为类变量（static修饰）分配内存并设置该类变量的默认初始化值，即零值。在初始化时才赋值。
  - final static修饰的变量在编译时就会分配内存，准备阶段显式初始化
  - 不会为实例变量分配初始化，类变量会分配在方法区，实例变量会随着对象一起分配到Java堆中
- 解析：将常量池中的符号引用转换为直接引用的过程

3.初始化

执行类的构造器方法clinit（）过程，javac编译器自动收集类中所有类变量的赋值动作和静态代码块

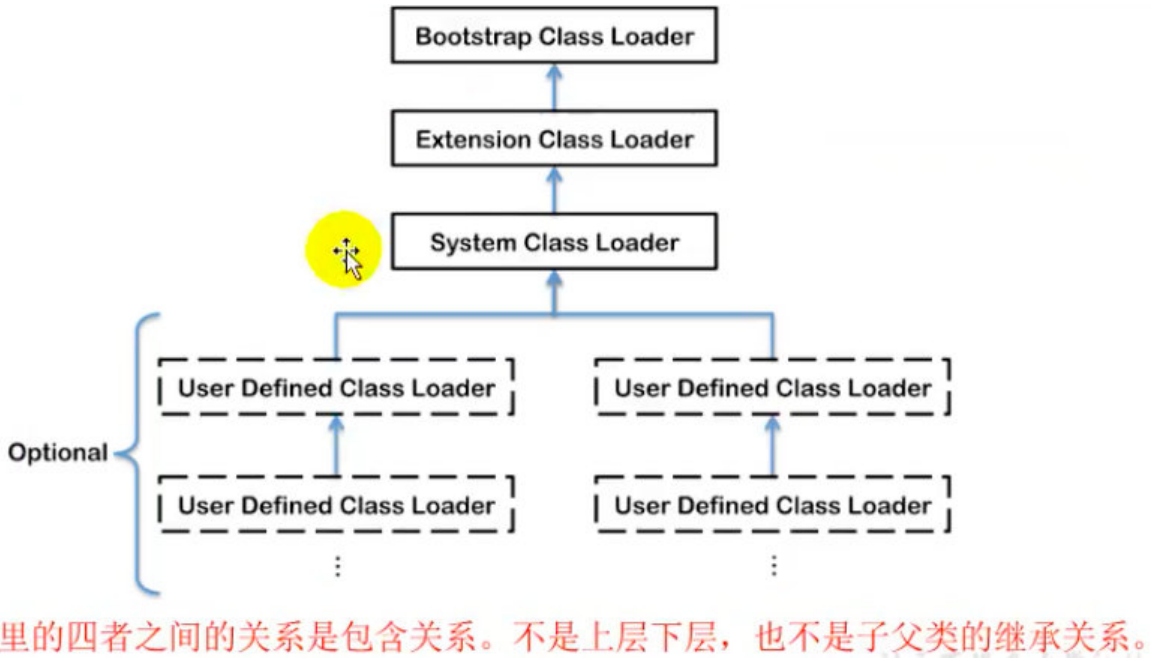
类加载器的分类

引导类加载器（Bootstrap ClassLoader）

使用c/c++实现，嵌套在JVM内部，没有父加载器，只加载包名为java，javax，sun等开头的类

自定义类加载器

指所有派生于抽象类ClassLoader的类加载器，由Java编写



这里的四者之间的关系是包含关系。不是上层下层，也不是子父类的继承关系。

- 拓展类加载器：ExtensionClassLoader
- 系统类加载器：AppClassLoader

Example：String类使用**引导类加载器**进行加载，因为Java核心内库都是由BootstrapClassLoader负责加载

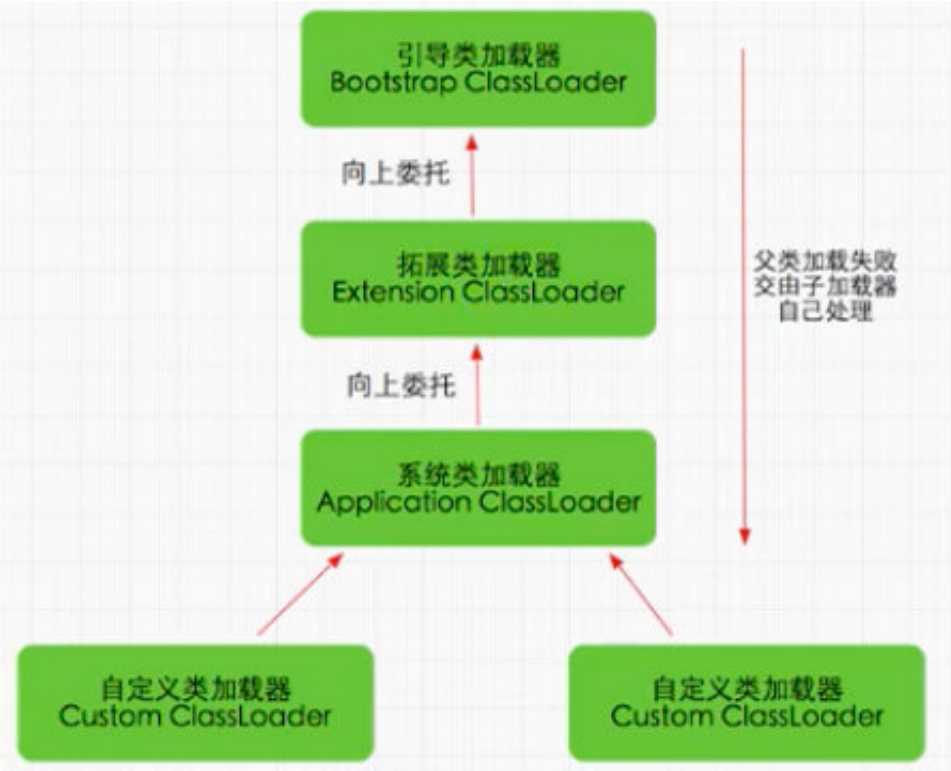
为什么要自定义类加载器?

1. 隔离加载类
2. 修改类加载的方式
3. 拓展加载源
4. 防止源码泄露

获取ClassLoader的途径

- 获取当前类的ClassLoader：class.getClassLoader()
- 获取当前线程上下文的ClassLoader：Thread.currentThread().getContextClassLoader()
- 获取系统的ClassLoader：ClassLoader.getSystemClassLoader()
- 获取调用者的ClassLoader：DriverManager.getCallerClassLoader()

双亲委派机制



工作流程：

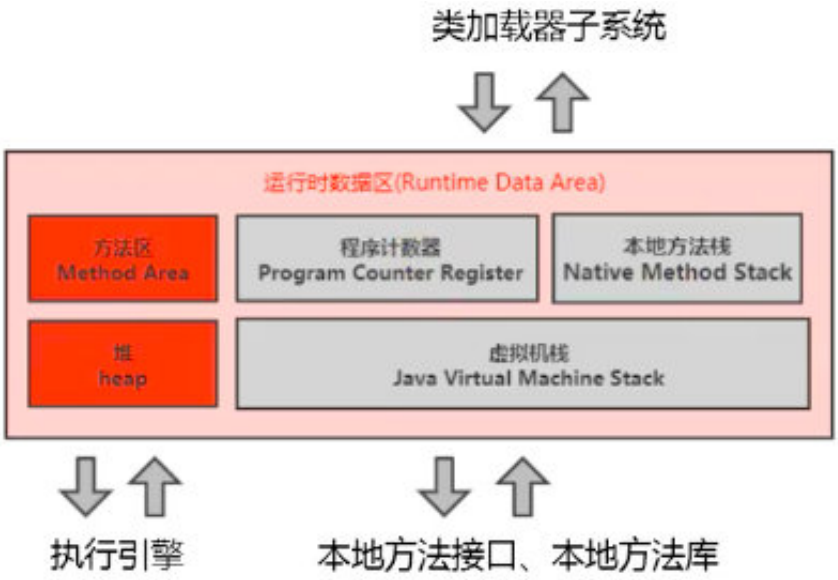
1. 如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行
2. 如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终到达顶层的引导类加载器
3. 如果父类加载器可以完成类加载任务，就成功返回；如无法完成，则子加载器才会尝试自己去加载

优势：

1. 避免类的重复加载
2. 保护程序安全，防止核心API被随意修改（沙箱安全机制）

Example：自定义类java.lang.String

运行时数据区



每个线程：独立包括程序计数器、虚拟机栈、本地方法栈

线程间共享：堆、本地方法区（或称堆外内存）

程序计数器（PC寄存器）

作用：用来存储指向下一条指令的地址，也就是即将要执行的指令代码，由执行引擎读取下一条指令

特点：即没有GC（GarbageCollection），也不会OOM（OutOfMemory）

使用PC寄存器存储字节码指令地址的作用?

因为CPU需要不停的切换各个线程，这时候切换回来以后，需要知道从哪开始继续执行。

JVM的字节码解释器（执行引擎）就需要通过改变PC寄存器的值来确定下一条应该执行什么样的字节码指令。

PC寄存器为什么会被设定为线程私有?

为了能够准确记录各个线程正在执行的当前字节码指令地址，最好的办法是为每一个线程都分配一个PC寄存器，这样各个线程之间便可以独立计算，从而不会出现相互干扰的情况。

由于CPU时间片轮限制，众多线程在**并发执行**过程中，任何一个确定的时刻，一个CPU或一个内核，只会执行某个线程中的一条指令

> 如何保证分毫不差?
>
> 每个线程在创建后，都会产生自己的PC寄存器和栈帧，PC寄存器在各个线程之间互不影响

虚拟机栈

> 内存中的栈和堆

> **栈是运行时的单位，而堆是存储的单位**

> To be specific，栈解决程序的运行问题，即程序如何执行，或者说如何处理数据。堆解决的是数据存储的问题，即数据怎么放，放在哪儿

Java虚拟机栈

每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（栈中存储数据的单位），对应着一次次Java方法的调用

- 是线程私有的
- 生命周期和线程一致
- 作用：主管Java程序的运行，它保存方法的局部变量（8种基本数据类型，对象的引用地址），部分结果，并参与方法的调用和返回

特点

- 栈是一种快速有效的分配存储方式，访问速度仅次于程序计数器
- 每个方法的执行，伴随着入栈、出栈操作
- 对于栈来说不存在垃圾回收问题（GC），但是存在OOM的问题

问题

1.开发中遇到的异常有哪些?

Java虚拟机规范允许Java栈的大小是固定不变的或者是动态的。

- 固定大小：超过容量，抛出StackOverFlowError异常
    - 设置栈的大小：-Xss1m
- 动态：在尝试拓展的时候无法申请到足够的内存，抛出OutOfMemoryError的异常

栈的存储单位

- 每个线程都有自己的栈，栈中的数据都是以**栈帧**的格式存在
- 在这个线程上正在执行的每个方法都各自对应一个栈帧（Stack Frame）
- 栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息

栈的执行原理

- 在一条活动线程中，一个时间点上，只会有一个活动的栈帧，即**只有当前正在执行的方法的栈帧是有效的**，这个栈帧被称为“当前栈帧”，与当前栈帧对应的方法就是“当前方法”，定义这个方法的类就是“当前类”
- 执行引擎运行的所有字节码指令只针对当前栈帧进行操作
- 如果该方法中调用了其他方法，对应的新的栈帧会被创建出来，放在栈的顶端，成为新的当前帧

栈帧的内部结构

五部分：局部变量表、操作数栈、动态链接、方法返回地址、一些附加信息

局部变量表

- 最基本的存储单元是Slot（变量槽）。32位以内的类型只占用一个slot（包括returnAddress类型），64位的类型（long和double）占用两个slot

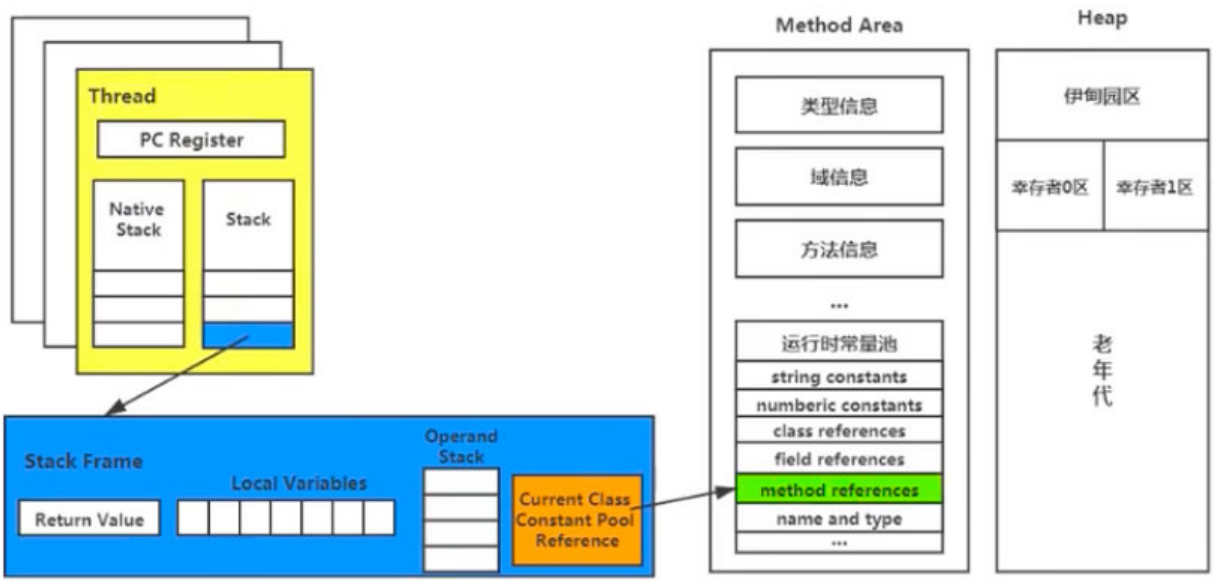
- 定义为一个**数字数组**（index由0到-1），主要用于存储方法参数和定义在方法体内的局部变量。这些数据类型包括各类基本数据类型、对象引用、returnAddress类型
- 由于局部变量表是建立在线程的栈上，是线程的私有数据，因此不存在数据安全问题
- 局部变量表所需的容量大小是编译期确定下来的，并保存在方法的Code属性的maximum local variables数据项中，在方法运行期间是不会改变局部变量表的大小的
- 如果当前帧是由构造方法或者实例方法创建的，那么该对象引用this将会放在index为0的slot处，其余的参数按照参数表顺序继续排列

操作数栈

在方法执行过程中，根据**字节码指令**，往栈中写入数据或提取数据。我们说Java虚拟机的解释引擎是基于栈的执行引擎，其中的栈就是操作数栈

- 主要用于保存计算过程中的中间结果，同时作为计算过程中变量临时的存储空间
- 数组实现。每一个操作数栈都会有一个明确的栈深度用于存储数值，其所需的最大深度在编译期就定义好了，保存在方法的Code属性中，为max_stack的值

动态链接



指向运行时常量池的方法引用

静态链接vs动态链接

- 静态链接（早期绑定）：被调用的方法在编译期可知
- 动态链接（晚期绑定）：被调用的方法在编译期无法确定

> 动态类型语言：判断变量值的类型信息，变量没有类型信息，变量值才有类型信息
>
> JS：var name = 'hello';
>
> Python：info = 12.1
>
> 静态类型语言：
>
> Java：String name = "world";

方法返回地址

存放调用该方法的pc寄存器的值（方法A调用方法B，当方法B执行完出栈，根据引用继续执行方法A）

一些附加信息

例如，对程序调试提供支持的信息

相关面试题

1. 举例栈溢出的情况

StackOverFlow，通过-Xss设置栈的大小：OOM

2. 调整栈大小，能保证不出现溢出嘛?

不能，举例：无穷循环

3. 分配的栈内存越大越好嘛?

不是，挤占其他空间

4. 垃圾回收是否会涉及到虚拟机栈?

不会，虚拟机栈只有OOM，没有GC

本地方法（Native Method）

定义：该方法的实现由非Java语言实现，用native关键字修饰

作用：融合不同的编程语言为Java所用，它的初衷是融合C/C++程序

本地方法栈（Native Method Stack）

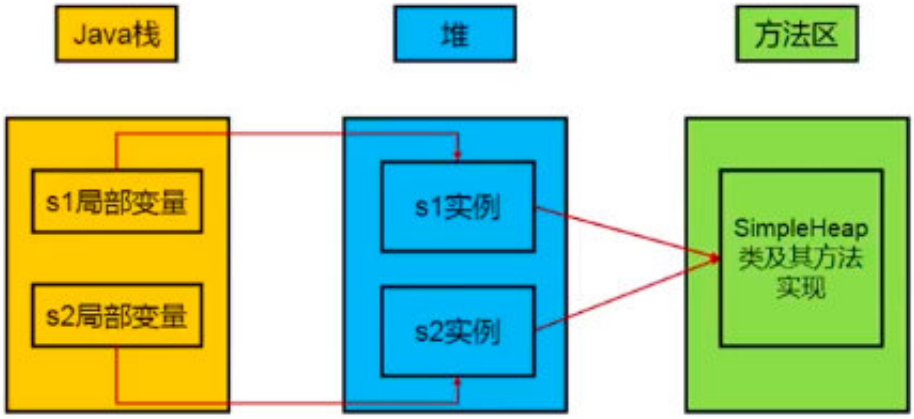
- Java虚拟机栈用于管理Java方法的调用，而本地方法栈用于管理本地方法的调用
- 允许被实现成固定或者是可动态拓展的内存大小
- 本地方法栈中登记本地方法，在执行引擎执行时加载本地方法库
- 在Hotspot JVM中，直接将本地方法栈和虚拟机栈合二为一

堆（Heap）

堆的概述

- 一个JVM实例只存在一个堆内存，堆也是Java内存管理的核心区域
- Java堆区在JVM启动时就被创建，其空间大小就确定了。是JVM管理的最大一块内存空间
- 堆可以处于物理上不连续的内存空间，但在逻辑上应该被视为连续的
- 所有的线程共享Java堆，在这里还可以划分线程私有的缓冲区（Thread Local Allocation Buffer，TLAB）

- 所有的对象实例和数组都应在运行时分配在堆上
- 数组和对象可能永远不会存储在栈上，因为栈帧中保存引用，这个引用指向对象或者数组中堆的位置



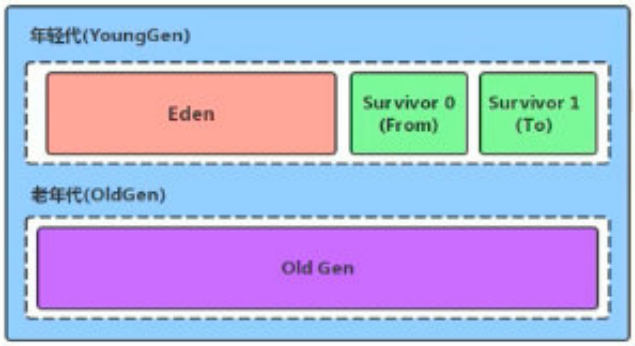
- 在方法结束后，堆中的对象不会马上被删除，仅仅在垃圾收集时才会被移除
- 堆，是GC执行垃圾回收的重点区域

堆的内存结构

- JDK7，堆内存逻辑上分为：新生区+养老区+永久区
- JDK8，堆内存逻辑上分为：新生区+养老区+元空间

> 约定：
>
> 新生区=新生代=年轻代
>
> 养老区=老年区=老年代
>
> 永久区=永久代

- 物理上，堆内存包括：年轻代（伊甸园区+幸存者0区+幸存者1区）+老年代



堆空间大小的设置

-Xms：用于表示堆区的起始内存

-Xmx：用于表示堆区的最大内存

- 一旦堆区中的内存大小超过-Xmx所指定的最大内存时，会抛出OutOfMemoryError异常
- **通常会将-Xms和-Xmx两个参数配置相同的值**，其目的是为了能够在java垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小，从而提高性能

> 查看设置的参数：
>
> 方式一：jps/jstat -gc 进程id
>
> 方式二：-XX:+PrintGCDetails

年轻代&老年代

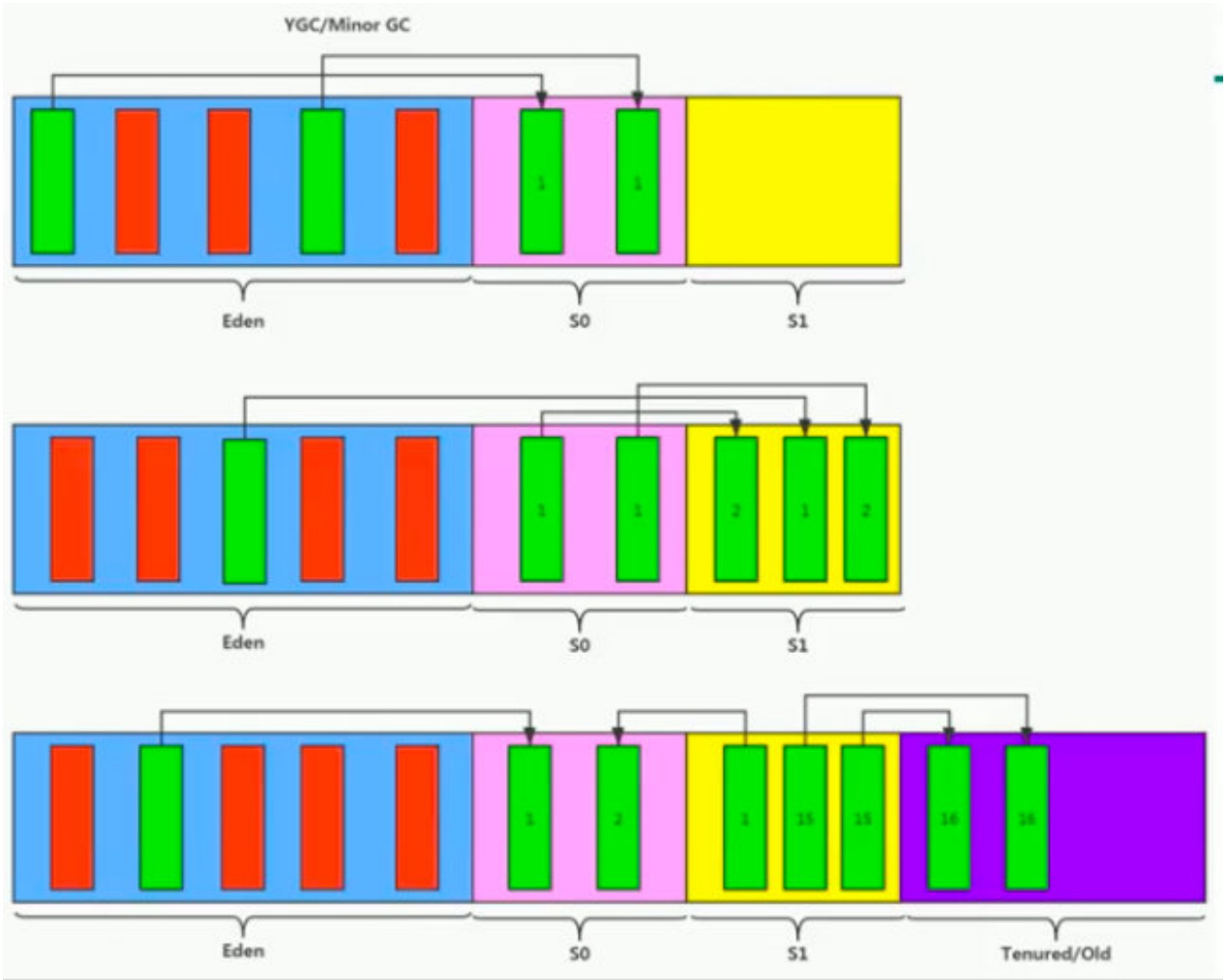
默认 -XX:NewRatio=2，表示年轻代：老年代=1:2，年轻代占整个堆的1/3

默认 -XX:SurvivorRatio=8，表示年轻代中Eden区与Survivor区的比例是8:1:1

- 几乎所有的Java对象都是在Eden区被new出来的
- 绝大部分的Java对象的销毁都在新生代进行了

-Xmn：用来设置年轻代的空间的大小（一般不设置）

对象分配的一般过程



阈值：-XX:MaxTenuringThreshold=进行设置，默认是15次

Minor GC & Major GC & Full GC

JVM在进行GC时，并非每次都对三个内存区域（新生代、老年代、方法区）一起回收的，大部分回收的都是指新生代

对于Hotspot VM，GC按照回收区域可以分为：

- 部分收集
  - 新生代收集（Minor GC/Young GC）：只是新生代（Eden、S0、S1）的垃圾收集
  - 老年代收集（Major GC/Old GC）：只是老年代的垃圾收集
  - 混合收集（Mixed GC）：收集整个新生代以及部分老年代的垃圾收集
- 整堆收集（Full GC）：收集整个Java堆和方法区的垃圾收集

Minor GC触发机制

- 当年轻代空间不足时，就会触发Minor GC，这里的年轻代满指的是Eden满，Survivor满不会触发GC（每次Minor GC会清理年轻代的内存）
- 因为Java对象大多具备**朝生夕灭**的特性，所以Minor GC非常频繁，一般回收速度也比较快
- Minor GC会引发STW，暂停其他的用户线程，等垃圾回收结束，用户线程才恢复运行

老年代GC（Major GC/Full GC）触发机制

- 出现了Major GC，经常会伴随至少一次的Minor GC（但并非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）
  - 老年代空间不足时，会先尝试触发Minor GC，如果之后空间还不足，则触发Major GC
- Major GC的速度一般会比Minor GC慢10倍以上，STW的时间更长
- 如果Major GC以后，内存还不足，就报OOM了

> Full GC是开发或调优中尽量要避免的，这样暂时时间会短一些

内存分配策略

- 优先分配到Eden
- 大对象直接分配到老年代，尽量避免出现过多的大对象
- 长期存活的对象分配到老年代
- 动态对象年龄判断

TLAB（Thread Local Allocation Buffer）

为什么有TLAB?

- 堆区是线程共享区域，任何线程都可以访问到堆区中的共享数据
- 由于对象实例的创建在JVM中非常频繁，因此在并发环境下从堆区中划分内存空间是线程不安全的
- 为避免多个线程操作同一地址，需要使用加锁等机制，从而影响分配速度

什么是TLAB?

- 对Eden区域继续进行划分，JVM为每个线程分配了一个私有缓存区域
- 多线程同时分配内存时，使用TLAB可以避免一系列的非线程安全问题，同时还能够提升内存分配的吞吐量
- 默认情况下，TLAB的内存非常小，仅占有整个Eden区的1%

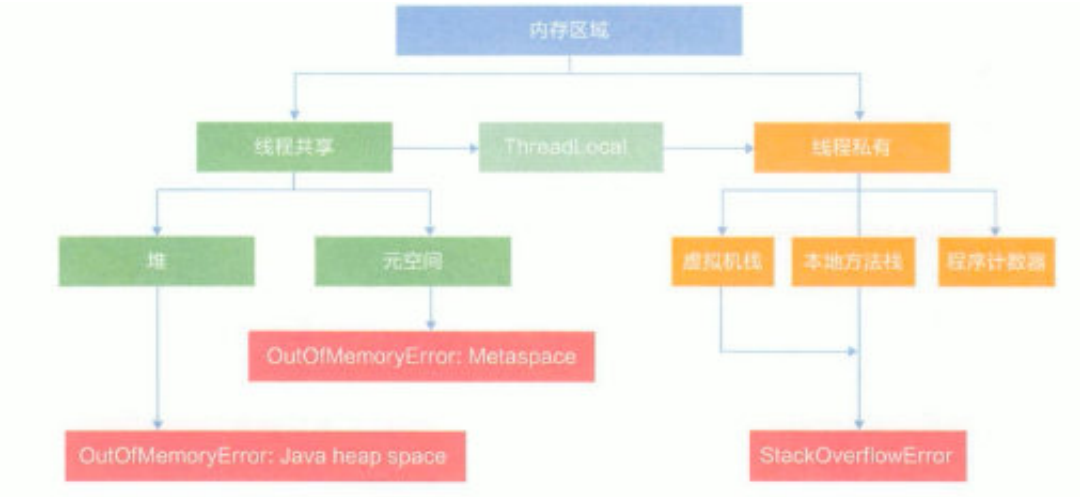
堆空间的参数设置

- -XX:+PrintFlagsInitial:查看所有参数的默认初始值
- -XX:+PrintFlagsFinal:查看所有的参数的最终值
- -Xms:初始堆内存空间
- -Xmx:最大堆空间内存
- -Xmn:设置新生代的大小
- -XX:NewRatio:配置新生代与老年代在堆结构中的占比
- -XX:SurvivorRatio:设置新生代中Eden和S0/S1空间的比例
- -XX:MaxTenuringThreshold:设置新生代垃圾的最大年龄
- -XX:+PrintGCDetails:输出详细的GC处理日志
- -XX:HandlePromotionFailure:是否设置空间分配担保

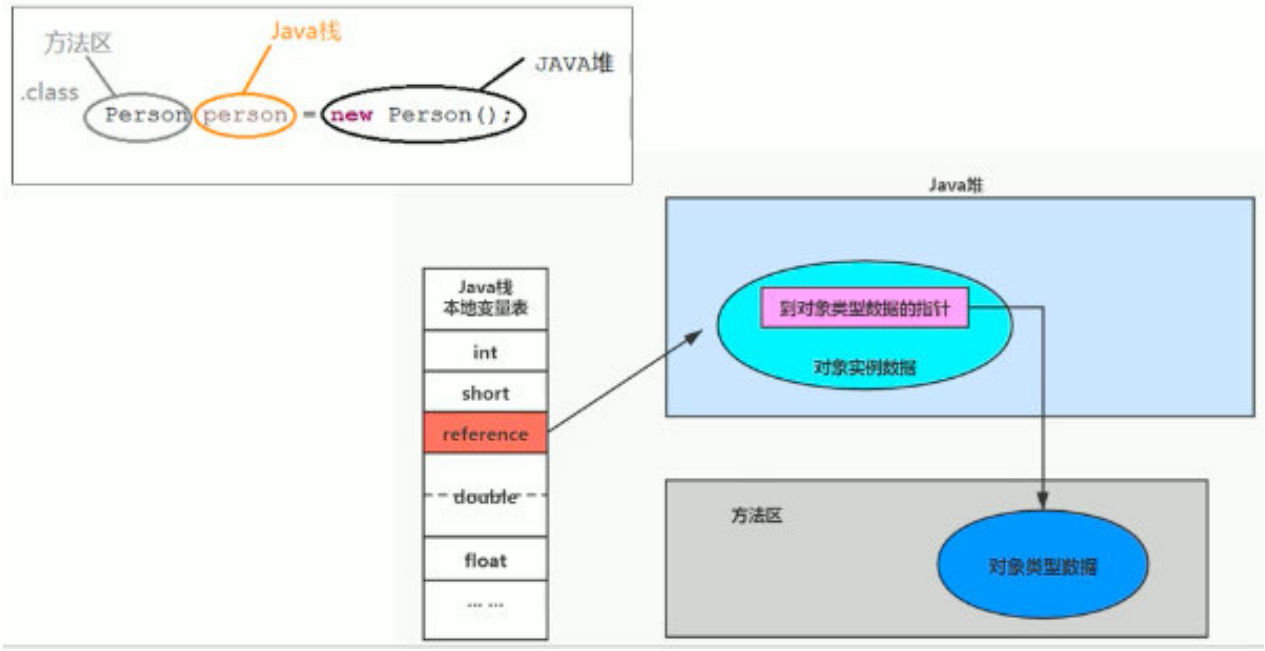
方法区

方法区&栈&堆的关系

从线程共享的角度来看



三者的交互关系



概念

方法区就是我们常说的**永久代**(Permanent Generation)**, 用于存储**被 **JVM 加载的类信息**、**常量**、**静态变量**、**即时编译器编译后的代码**等数据.

HotSpot VM 把 GC 分代收集扩展至方法区, 即**使用 Java 堆的永久代来实现方法区**, 这样 HotSpot 的垃圾收集器就可以像管理 Java 堆一样管理这部分内存, 而不必为方法区开发专门的内存管理器(永久带的内存回收的主要目标是针对**常量池的回收**和**类型 的卸载**, 因此收益一般很小)。

运行时常量池

运行时常量池(Runtime Constant Pool)是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量池(Constant Pool Table)，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

Java 虚拟机对 Class 文件的每一部分(自然也包括常量池)的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会 被虚拟机认可、装载和执行。