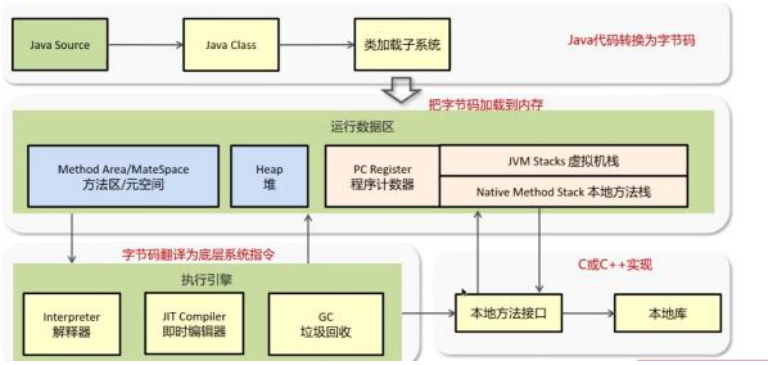
JVM组成、类加载器

2024年3月30日 22:12

JVM由哪些部分组成，运行流程是什么?

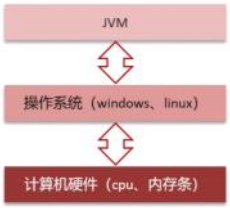


JVM是什么

Java Virtual Machine Java程序的运行环境（java二进制字节码的运行环境）

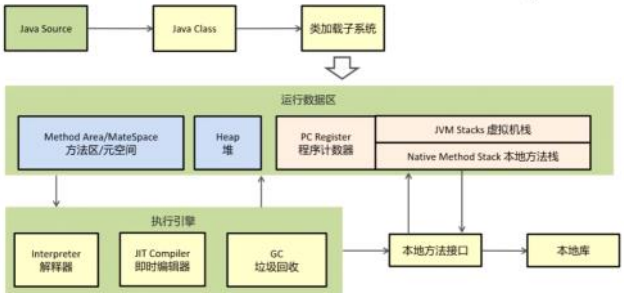
好处:

- 一次编写，到处运行
- 自动内存管理，垃圾回收机制



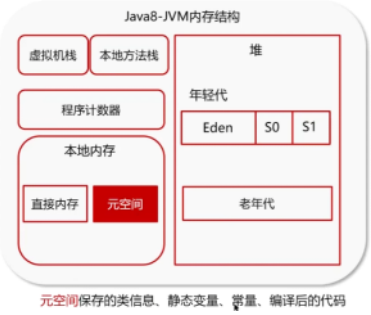
什么是程序计数器?

程序计数器：线程私有的，内部保存的字节码的行号。用于记录正在执行的字节码指令的地址。



你能给我详细的介绍Java堆吗?

线程共享的区域：主要用来保存对象实例，数组等，当堆中没有内存空间可分配给实例，也无法再扩展时，则抛出OutOfMemoryError异常。



元空间保存的类信息、静态变量、常量、编译后的代码

年轻代被划分为三部分，Eden区和两个大小严格相同的Survivor区，根据JVM的策略，在经过几次垃圾收集后，任然存活于Survivor的对象将被移动到老年代区间。

老年代主要保存生命周期长的对象，一般是一些老的对象

你能给我详细的介绍Java堆吗?

线程共享的区域：主要用来保存对象实例，数组等，当堆中没有内存空间可分配给实例，也无法再扩展时，则抛出OutOfMemoryError异常。

方法区和永久代以及元空间是什么关系呢？ 方法区和永久代以及元空间的关系很像 Java 中接口和类的关系，类实现了接口，这里的类就可以看作是永久代和元空间，接口可以看作是方法区，也就是说永久代以及元空间是 HotSpot 虚拟机对虚拟机规范中方法区的两种实现方式。并且，永久代是 JDK 1.8 之前的方法区实现，JDK 1.8 及以后方法区的实现变成了元空间。

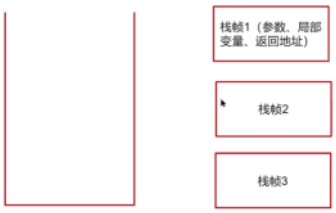
你能给我详细的介绍Java堆吗?

- 线程共享的区域：主要用来保存对象实例，数组等，内存不够则抛出OutOfMemoryError异常。
- 组成：年轻代+老年代
  - 年轻代被划分为三部分，Eden区和两个大小严格相同的Survivor区
  - 老年代主要保存生命周期长的对象，一般是一些老的对象
- Jdk1.7和1.8的区别
  - 1.7中有有一个永久代，存储的是类信息、静态变量、常量、编译后的代码
  - 1.8移除了永久代，把数据存储到了本地内存的元空间中，防止内存溢出

什么是虚拟机栈

Java Virtual machine Stacks (java 虚拟机栈)

- 每个线程运行时所需要的内存，称为虚拟机栈，先进后出
- 每个栈由多个栈帧（frame）组成，对应着每次方法调用时所占用的内存
- 每个线程只能有一个活动栈帧，对应着当前正在执行的那个方法



什么是虚拟机栈

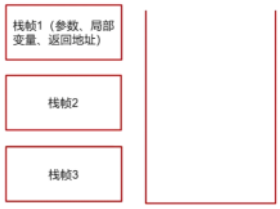
1. 垃圾回收是否涉及栈内存?

   垃圾回收主要指就是堆内存，当栈帧弹栈以后，内存就会释放

2. 栈内存分配越大越好吗?

   未必，默认的栈内存通常为1024k

   栈帧过大会导致线程数变少，例如，机器总内存为512m，目前能活动的线程数则为512个，如果把栈内存改为2048k，那么能活动的栈帧就会减半



什么是虚拟机栈

3. 方法内的局部变量是否线程安全?

- 如果方法内局部变量没有逃离方法的作用范围，它是线程安全的
- 如果是局部变量引用了对象，并逃离方法的作用范围，需要考虑线程安全

```
public static void main(String[] args) {
    StringBuilder sb = new StringBuilder();
    sb.append(1);
    sb.append(2);
    new Thread(()->{
        m2(sb);
    }).start();
}
public static void m1(){
    StringBuilder sb = new StringBuilder();
    sb.append(1);
    sb.append(2);
    System.out.println(sb.toString());
}
public static void m2(StringBuilder sb){
    sb.append(3);
    sb.append(4);
    System.out.println(sb.toString());
}
public static StringBuilder m3(){
    StringBuilder sb = new StringBuilder();
    sb.append(5);
    sb.append(6);
    return sb;
}
```

线程安全

线程不安全

线程不安全

栈内存溢出情况

- 栈帧过多导致栈内存溢出，典型问题：递归调用
- 栈帧过大导致栈内存溢出

```
public static void m4(){
  m4();
}
```

java.lang.StackOverflowError

6.堆栈的区别是什么？

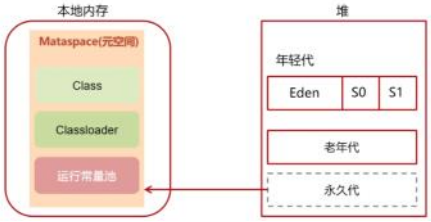
- 栈内存一般会用来存储局部变量和方法调用，但堆内存是用来存储Java对象和数组的的。堆会GC垃圾回收，而栈不会。
- 栈内存是线程私有的，而堆内存是线程共有的。
- 两者异常错误不同，但如果栈内存或者堆内存不足都会抛出异常。

  栈空间不足：java.lang.StackOverFlowError。

  堆空间不足：java.lang.OutOfMemoryError。

能不能解释一下方法区？

- 方法区(Method Area)是各个线程共享的内存区域
- 主要存储类的信息、运行时常量池
- 虚拟机启动的时候创建，关闭虚拟机时释放
- 如果方法区域中的内存无法满足分配请求，则会抛出OutOfMemoryError: Metaspace



常量池

可以看作是一张表，虚拟机指令根据这张常量表找到要执行的类名、方法名、参数类型、字面量等信息

```
javap -v Application.class
```

查看字节码结构（类的基本信息、常量池、方法定义）

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #2
       3: ldc           #3
       5: invokevirtual #4
       8: return
```

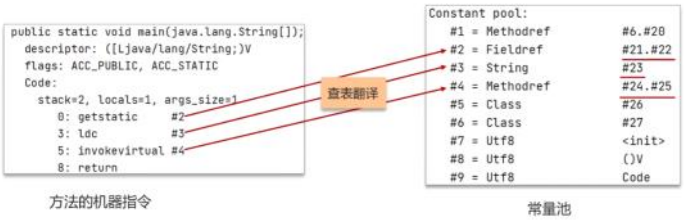
方法的机器指令

```
Constant pool:
   #1 = Methodref          #6.#20
   #2 = Fieldref           #21.#22
   #3 = String             #23
   #4 = Methodref          #24.#25
   #5 = Class              #26
   #6 = Class              #27
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
```

常量池

运行时常量池

常量池是 *.class 文件中的，当该类被加载，它的常量池信息就会放入运行时常量池，并把里面的符号地址变为真实地址



方法的机器指令　　常量池

1.能不能解释一下方法区？

- 方法区(Method Area)是各个线程共享的内存区域
- 主要存储类的信息、运行时常量池
- 虚拟机启动的时候创建，关闭虚拟机时释放
- 如果方法区域中的内存无法满足分配请求，则会抛出OutOfMemoryError: Metaspace

2.介绍一下运行时常量池

- 常量池：可以看作是一张表，虚拟机指令根据这张常量表找到要执行的类名、方法名、参数类型、字面量等信息
- 当类被加载，它的常量池信息就会放入运行时常量池，并把里面的符号地址变为真实地址

你听过直接内存吗？

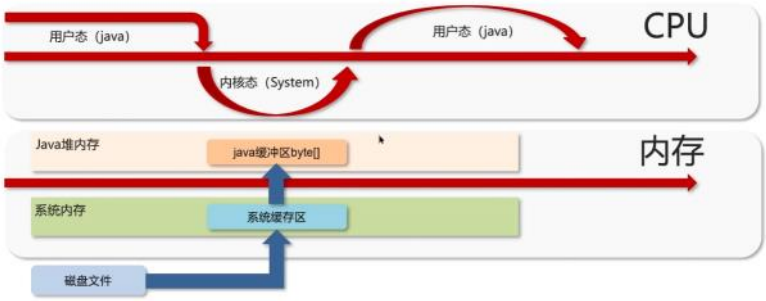
直接内存：并不属于JVM中的内存结构，不由JVM进行管理。是虚拟机的系统内存，常见于 NIO 操作时，用于数据缓冲区，它分配回收成本较高，但读写性能高

举例

Java代码完成文件拷贝

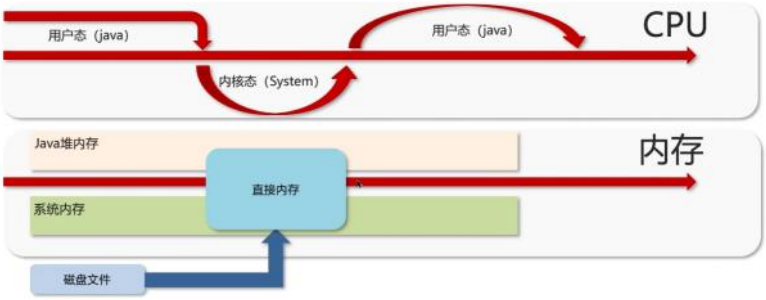
你听过直接内存吗？

常规IO的数据拷贝流程



你听过直接内存吗？

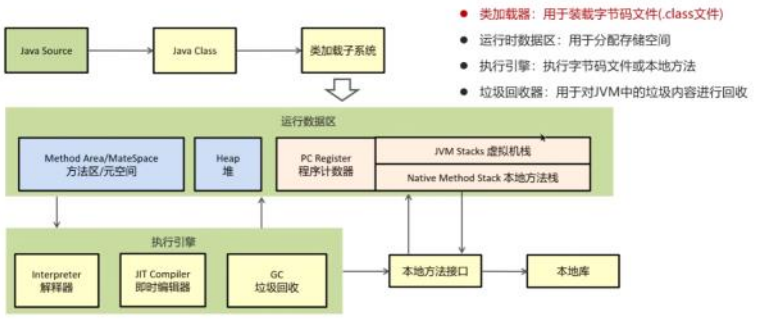
NIO数据拷贝流程



你听过直接内存吗？

- 并不属于JVM中的内存结构，不由JVM进行管理。是虚拟机的系统内存
- 常见于 NIO 操作时，用于数据缓冲区，分配回收成本较高，但读写性能高，不受 JVM 内存回收管理

什么是类加载器，类加载器有哪些

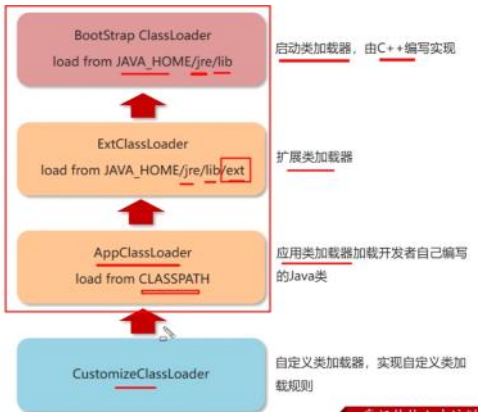


- 类加载器：用于装载字节码文件(.class文件)
- 运行时数据区：用于分配存储空间
- 执行引擎：执行字节码文件或本地方法
- 垃圾回收器：用于对JVM中的垃圾内容进行回收

什么是类加载器，类加载器有哪些

类加载器

JVM只会运行二进制文件，类加载器的作用就是将字节码文件加载到JVM中，从而让Java程序能够启动起来。



1. 什么是类加载器

JVM只会运行二进制文件，类加载器的作用就是将字节码文件加载到JVM中，从而让Java程序能够启动起来。

2. 类加载器有哪些

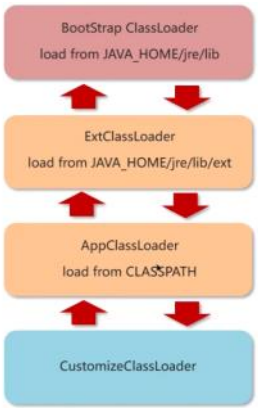
- 启动类加载器(BootStrap ClassLoader):加载JAVA_HOME/jre/lib目录下的库
- 扩展类加载器(ExtClassLoader):主要加载JAVA_HOME/jre/lib/ext目录中的类
- 应用类加载器(AppClassLoader):用于加载classPath下的类
- 自定义类加载器(CustomizeClassLoader):自定义类继承ClassLoader，实现自定义类加载规则。

什么是双亲委派模型?

加载某一个类，先委托上一级的加载器进行加载，如果上级加载器也有上级，则会继续向上委托，如果该类委托上级没有被加载，子加载器尝试加载该类

Student类

String类



JVM为什么采用双亲委派机制?

(1) 通过双亲委派机制可以避免某一个类被重复加载，当父类已经加载后则无需重复加载，保证唯一性。

(2) 为了安全，保证类库API不会被修改

```
package java.lang;

public class String {

    public static void main(String[] args) {
        System.out.println("demo info");
    }
}
```

由于是双亲委派的机制，java.lang.String的在启动类加载器得到加载，因为在核心jre库中有其相同名字的类文件，但该类中并没有main方法。这样就能防止恶意篡改核心API库。

此时执行main函数，会出现异常，在类 java.lang.String 中找不到 main 方法

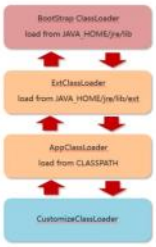
```
错误: 在类 java.lang.String 中找不到 main 方法, 请将 main 方法定义为:
   public static void main(String[] args)
否则 JavaFX 应用程序类必须扩展javafx.application.Application
```

1. 什么是双亲委派模型?

加载某一个类，先委托上一级的加载器进行加载，如果上级加载器也有上级，则会

继续向上委托，如果该类委托上级没有被加载，子加载器尝试加载该类

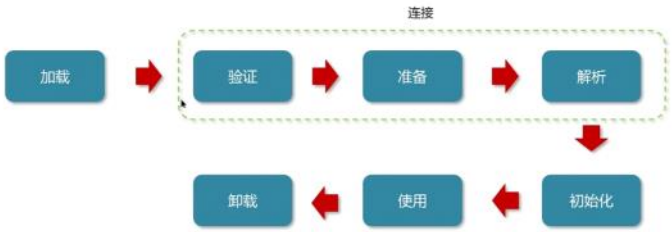
2. JVM为什么采用双亲委派机制?



- 通过双亲委派机制可以避免某一个类被重复加载，当父类已经加载后则无需重复加载，保证唯一性。
- 为了安全，保证类库API不会被修改

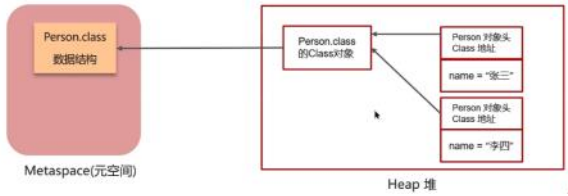
说一下类装载的执行过程?

类从加载到虚拟机中开始，直到卸载为止，它的整个生命周期包括了：加载、验证、准备、解析、初始化、使用和卸载这7个阶段。其中，验证、准备和解析这三个部分统称为连接（linking）



加载

- 通过类的全名，获取类的二进制数据流。
- 解析类的二进制数据流为方法区内的数据结构（Java类模型）
- 创建java.lang.Class类的实例，表示该类型。作为方法区这个类的各种数据的访问入口



验证

验证类是否符合 JVM规范，安全性检查

```
Constant pool:
   #1 = Methodref          #6.#20
   #2 = Fieldref           #21.#22
   #3 = String             #23
   #4 = Methodref          #24.#25
   #5 = Class              #26
   #6 = Class              #27
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
```

(1)文件格式验证
(2)元数据验证
(3)字节码验证
格式检查，如：文件格式是否错误、语法是否错误、字节码是否合规

(4)符号引用验证　Class文件在其常量池会通过字符串记录自己将要使用的其他类或者方法，检查它们是否存在

准备

为类变量分配内存并设置类变量初始值

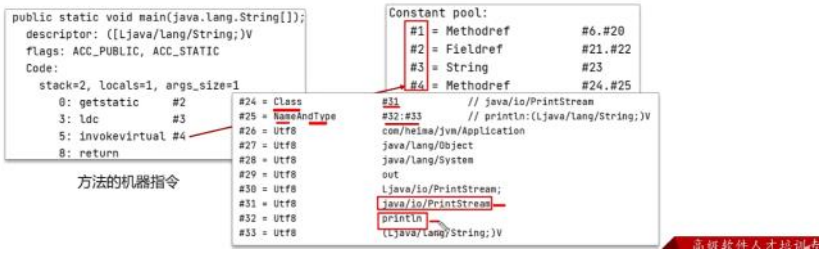
- static变量，分配空间在准备阶段完成（设置默认值），赋值在初始化阶段完成
- static变量是final的基本类型，以及字符串常量，值已确定，赋值在准备阶段完成
- static变量是final的引用类型，那么赋值也会在初始化阶段完成

```
public class Application {

    static int b = 10;
    static final int c = 20;
    static final String d = "hello";
    static final Object obj = new Object();

}
```

解析

把类中的符号引用转换为直接引用

比如：方法中调用了其他方法，方法名可以理解为符号引用，而直接引用就是使用指针直接指向方法。



方法的机器指令

初始化

对类的静态变量，静态代码块执行初始化操作

- 如果初始化一个类的时候，其父类尚未初始化，则优先初始化其父类。
- 如果同时包含多个静态变量和静态代码块，则按照自上而下的顺序依次执行。

初始化

JVM 开始从入口方法开始执行用户的程序代码

- 调用静态类成员信息（比如：静态字段、静态方法）
- 使用new关键字为其创建对象实例

总结

说一下类装载的执行过程？

- 加载:查找和导入class文件
- 验证:保证加载类的准确性
- 准备:为类变量分配内存并设置类变量初始值
- 解析:把类中的符号引用转换为直接引用
- 初始化:对类的静态变量，静态代码块执行初始化操作
- 使用:JVM 开始从入口方法开始执行用户的程序代码
- 卸载:当用户程序代码执行完毕后，JVM便开始销毁创建的Class对象。