

jvm

- jvm
 - 类加载子系统
 - 1.类加载子系统作用
 - 1.1类加载器ClassLoader角色
 - 1.2加载
 - 1.3 链接
 - 1.3.1 验证:
 - 1.3.2 准备
 - 1.3.3 解析
 - 1.4初始化
 - 2.类加载器分类
 - 2.1 自定义类与核心类库的加载器
 - 2.2 虚拟机自带的加载器
 - 2.3 用户自定义类加载器
 - 3 ClassLoader的常用方法及获取方法
 - 3.1 ClassLoader类，它是一个抽象类，其后所有的类加载器都继承自ClassLoader（不包括启动类加载器）
 - 3.2 ClassLoader继承关系
 - 3.3 获取ClassLoader的途径
 - 4. 双亲委派机制
 - 4.1 双亲委派机制工作原理
 - 4.2 双亲委派机制的优势
 - 5. 沙箱安全机制
 - 6.其他
 - 对类加载器的引用
 - 类的主动使用和被动使用
 - 运行时数据区1-[程序计数器+虚拟机栈+本地方法栈]
 - 内存与线程
 - 1 内存
 - 2 分区介绍
 - 3 线程
 - 3.1 JVM系统线程
 - 1.程序计数器（PC寄存器）
 - 1.1 作用
 - 1.2 代码示例

- 1.3 面试常问
- 2.虚拟机栈
 - 2.1概述
 - 2.1.1 背景
 - 2.1.2 内存中的堆与栈
 - 2.1.3 虚拟机栈是什么
 - 2.1.4 栈的特点
 - 2.1.5 栈中可能出现的异常
 - 2.1.6设置栈的内存大小
 - 2.2 栈的存储结构和运行原理
 - 2.2.1
 - 2.2.2 栈帧的内部结构
 - 2.3 局部变量表（Local Variables）
 - 2.3.1 概述
 - 2.3.2 变量槽slot的理解与演示
 - 2.3.3 slot的重复利用
 - 2.3.4 静态变量与局部变量的对比及小结
 - 2.4 操作数栈（Operand Stack）
 - 2.4.1 概述
 - 2.4.2 i++ 和 ++i的区别
 - 2.4.3 栈顶缓存技术ToS（Top-of-Stack Cashing）
 - 2.5 动态链接（Dynamic Linking）
 - 2.5.1方法的调用
 - 2.5.2虚方法和非虚方法
 - 2.5.3方法重写的本质
 - 2.5.4 虚方法表
 - 2.6 方法返回地址（Return Address）
- 3.本地方法栈
- 运行时数据区2-堆
 - 1.核心概述
 - 1.1 配置jvm及查看jvm进程
 - 1.2 分析SimpleHeap的jvm情况
 - 1.3 堆的细分内存结构
 - 2.设置堆内存大小与OOM
 - 2.1 查看堆内存大小
 - 2.2 堆大小分析
 - 3.年轻代与老年代
 - 4.图解对象分配过程
 - 4.1

- 4.2 对象分配的特殊情况
- 5.Minor GC、Major GC、Full GC
- 6.内存分配策略
- 7.为对象分配内存: TLAB (线程私有缓存区域)
- 8.小结堆空间的参数设置
- 9.堆是分配对象的唯一选择么 (不是)
- 运行时数据区3-方法区
 - 1. 堆、栈、方法区的交互关系
 - 2. 方法区的理解
 - 3. 设置方法区大小与OOM
 - 4.方法区的内部结构
 - 类型信息
 - 域信息 (成员变量)
 - 方法信息
 - 运行时常量池
 - 常量池
 - 运行时常量池
 - 5.方法区的使用举例
 - 6.方法区的演进细节
 - 7.方法区的垃圾回收
 - 8.总结
- 运行时数据区4-对象的实例化内存布局与访问定位+直接内存
 - 1.对象的实例化
 - 1.1 创建对象的方式
 - 1.2 创建对象的步骤
 - 1) 判断对象对应的类是否加载、链接、初始化
 - 2) 为对象分配内存
 - 3) 处理并发安全问题
 - 4) 初始化分配到的空间
 - 5) 设置对象的对象头
 - 6) 执行init方法进行初始化
 - 代码示例
 - 2.对象的内存布局
 - 对象头 (Header)
 - 实例数据 (Instance Data)
 - 对齐填充 (Padding)
 - 小结
 - 3.对象的访问定位
- 执行引擎 (Execution Engine)

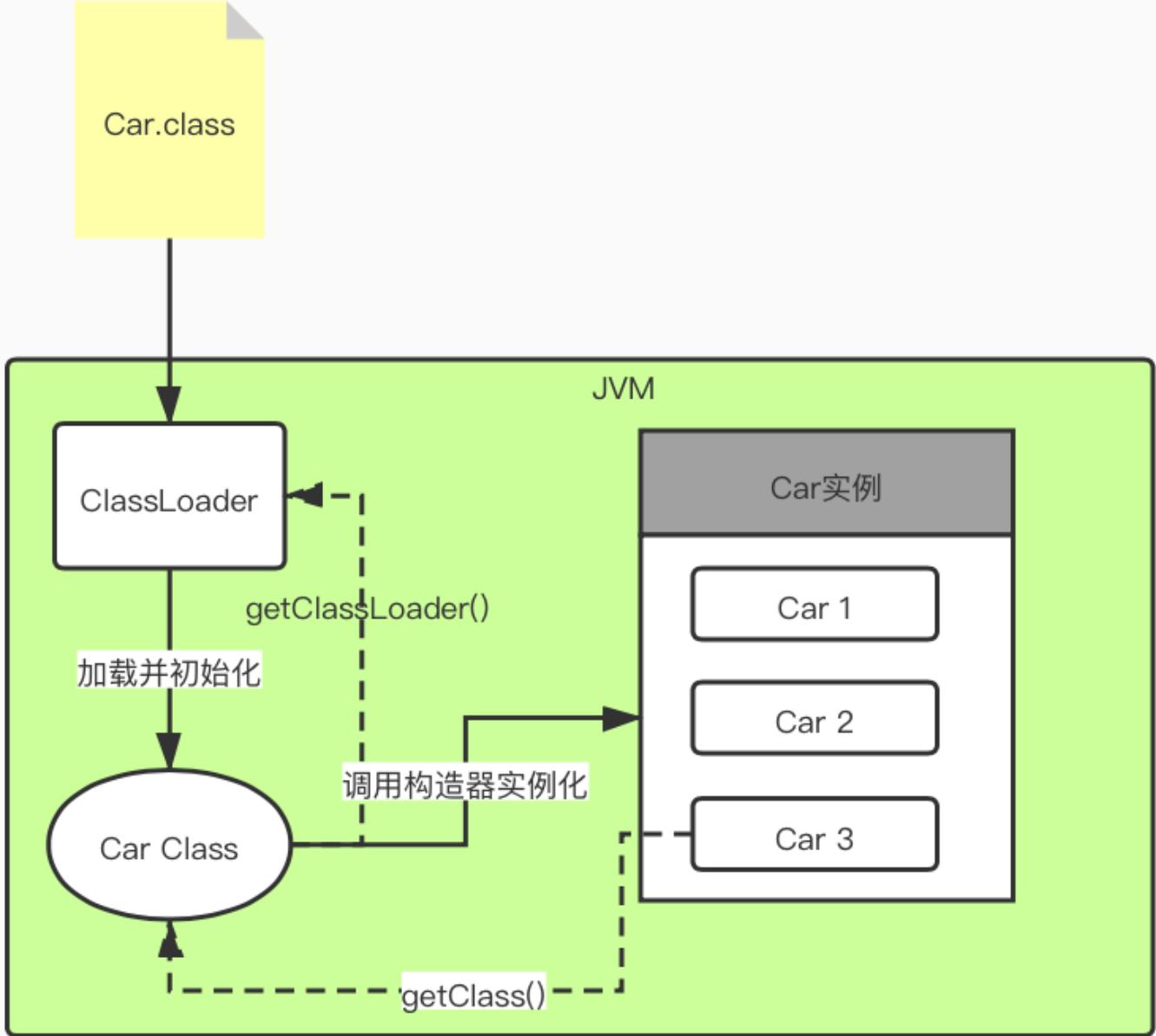
- 执行引擎概述
- Java代码编译和执行过程
- 解释器

类加载子系统

1.类加载子系统作用

- 类加载子系统负责从文件系统或者网络中加载Class文件，class文件在文件开头有特定的文件标识；
- ClassLoader只负责class文件的加载，至于它是否可以运行，则由Execution Engine决定
- 加载的类信息存放于一块称为方法区的内存空间。除了类信息之外，方法区还会存放运行时常量池信息，可能还包括字符串字面量和数字常量^[1]（这部分常量信息是Class文件中常量池部分的内存映射）

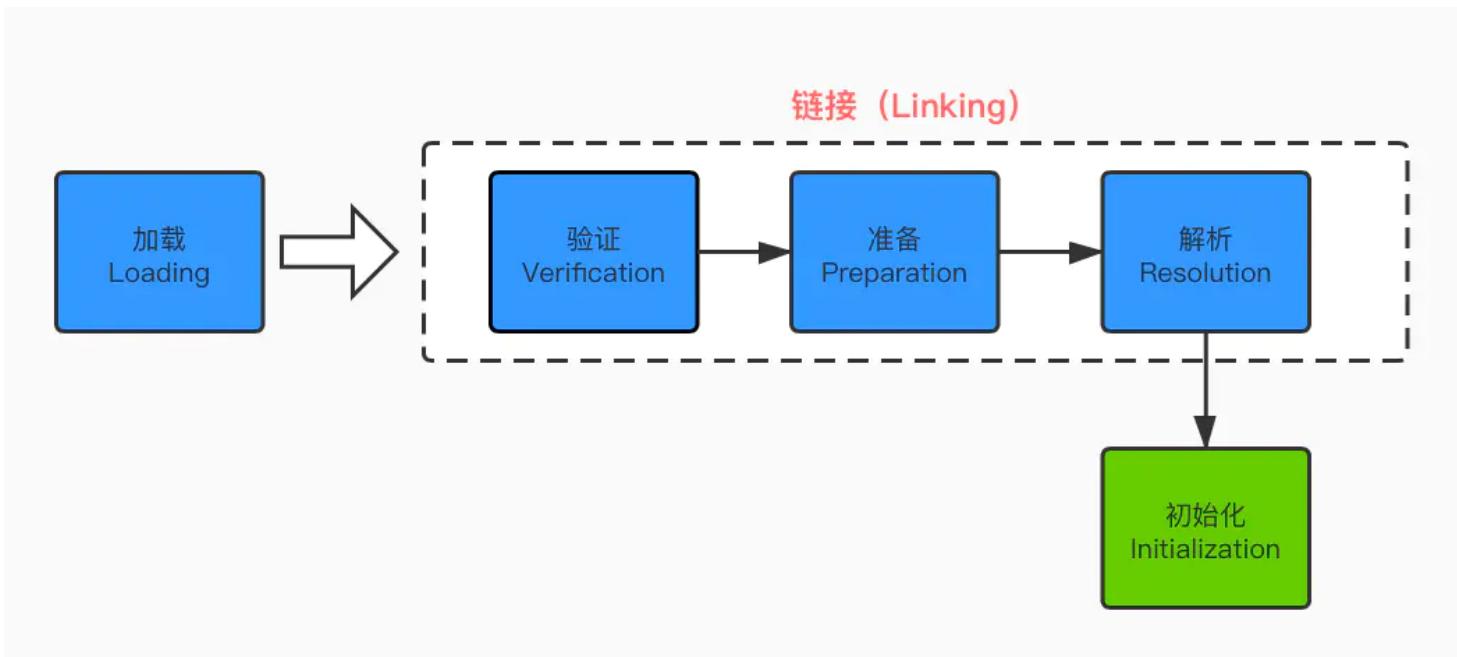
1.1类加载器ClassLoader角色



1.2 加载

1. 通过一个类的全限定名获取定义此类的二进制字节流；
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据；
3. 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

1.3 链接



1.3.1 验证:

- 目的在于确保**Class**文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不会危害虚拟机自身安全。
- 主要包括四种验证，文件格式验证，源数据验证，字节码验证，符号引用验证。

1.3.2 准备

- 为类变量(静态变量)分配内存并且设置该类变量的默认初始值，即零值；
- 这里不包含用**final**修饰的常量，因为**final**在编译的时候就会分配了，准备阶段会显式初始化；
- 这里不会为实例变量分配初始化，类变量(静态变量)会分配在方法区中，而实例变量是会随着对象一起分配到**java**堆中。

1.3.3 解析

- 将常量池内的符号引用转换为直接引用的过程。
- 事实上，解析操作会伴随着**jvm**在执行完初始化之后再执行
- 符号引用就是一组符号来描述所引用的目标。符号应用的字面量形式明确定义在《**java**虚拟机规范》的**class**文件格式中。直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄
- 解析动作主要针对类或接口、字段、类方法、接口方法、方法类型等。对应常量池中的 **CONSTANT_Class_info / CONSTANT_Fieldref_info / CONSTANT_Methodref_info** 等。

1.4 初始化

- 初始化阶段就是执行类构造器方法 **cinit()** 的过程。
- 此方法不需要定义，是**javac**编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语句合并而来。我们注意到如果没有静态变量**c**，那么字节码文件中就不会有**cinit**方法

```

    /**
     * 类的初始化 <clinit>
     */
    public class ClassInitTest {
        //任何一个类声明以后，内部至少存在一个类的构造器
        private int a = 1;
        //    private static int c = 3;

        public static void main(String[] args) {
            int b = 2;
        }
    }

```

```

    /**
     * 类的初始化 <clinit>
     */
    public class ClassInitTest {

        private int a = 1;
        private static int c = 3;

        public static void main(String[] args) {
            int b = 2;
        }
    }

```

- 构造器方法中指令按语句在源文件中出现的顺序执行

```

    /**
     * 类的初始化 <clinit>
     */
    public class ClassInitTest {
        private static int num = 1;

        static {
            num = 2;
            number = 20;//可以赋值
            System.out.println(num);
            System.out.println(number);//报错：非法前向引用，不能调用
        }

        private static int number = 10;//LinkingZprepare number =0;-->initial:覆盖 20 -->10

        public static void main(String[] args) {
            System.out.println(ClassInitTest.num);//2
            System.out.println(ClassInitTest.number);//10
        }
    }

```

- clinit()不同于类的构造器。（关联：构造器是虚拟机视角下的init()）
- 若该类具有父类，jvm会保证子类的clinit()执行前，父类的clinit()已经执行完毕

```

public class ClinitTest1 {
    static class Father{
        public static int A = 1;
        static {
            A = 2;
        }
    }

    static class Son extends Father{
        public static int B = A;
    }

    public static void main(String[] args) {
        //加载Father类，其次加载Son类
        System.out.println(Son.B);
    }
}

```

- 虚拟机必须保证一个类的clinit()方法在多线程下被同步加锁。

```
public class DeadThreadTest {
    public static void main(String[] args) {
        Runnable runnable = ()->{
            System.out.println(Thread.currentThread().getName()+"开始");
            DeadThread thread = new DeadThread();
            System.out.println(Thread.currentThread().getName()+"结束");
        };

        Thread thread1 = new Thread(runnable, name: "线程1");
        Thread thread2 = new Thread(runnable, name: "线程2");
        thread1.start();
        thread2.start();
    }
}

class DeadThread{
    static {
        if (true){
            System.out.println(Thread.currentThread().getName()+"初始化当前类");
            while (true){
            }
        }
    }
}
```

DeadThreadTest > main()

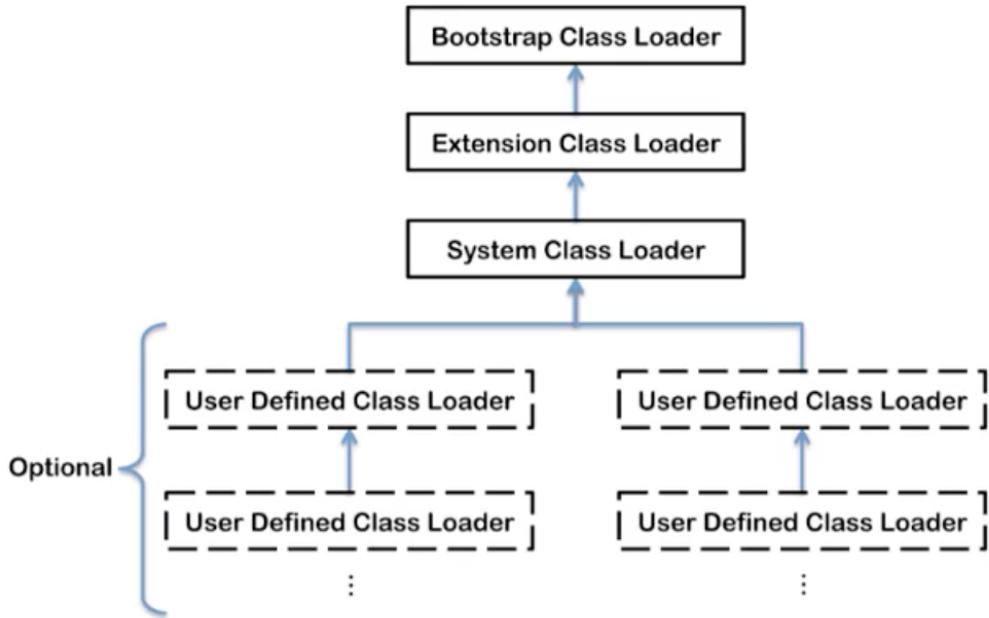
DeadThreadTest

/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin/java ...

线程1开始
线程2开始
线程1初始化当前类

2.类加载器分类

- JVM支持两种类型的加载器，分别为引导类加载器（BootStrap ClassLoader）和自定义类加载器（User-Defined ClassLoader）
- 从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是java虚拟机规范却没有这么定义，而是将所有派生于抽象类ClassLoader的类加载器都划分为自定义类加载器。
- 无论类加载器的类型如何划分，在程序中我们最常见的类加载器始终只有三个，如下所示：



这里的四者之间的关系是包含关系。不是上层下层，也不是父子类的继承关系。
让天下没有难学的技术

2.1 自定义类与核心类库的加载器

- 对于用户自定义类来说：使用系统类加载器AppClassLoader进行加载
- java核心类库都是使用引导类加载器BootStrapClassLoader加载的

```

/**
 * ClassLoader加载
 */
public class ClassLoaderTest {
    public static void main(String[] args) {
        //获取系统类加载器
        ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
        System.out.println(systemClassLoader); //sun.misc.Launcher$AppClassLoader@18b4aac2

        //获取其上层 扩展类加载器
        ClassLoader extClassLoader = systemClassLoader.getParent();
        System.out.println(extClassLoader); //sun.misc.Launcher$ExtClassLoader@610455d6

        //获取其上层 获取不到引导类加载器
        ClassLoader bootStrapClassLoader = extClassLoader.getParent();
        System.out.println(bootStrapClassLoader); //null

        //对于用户自定义类来说：使用系统类加载器进行加载
        ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
        System.out.println(classLoader); //sun.misc.Launcher$AppClassLoader@18b4aac2

        //String 类使用引导类加载器进行加载的 -->java核心类库都是使用引导类加载器加载的
        ClassLoader classLoader1 = String.class.getClassLoader();
        System.out.println(classLoader1); //null
    }
}

```

2.2 虚拟机自带的加载器

1. 启动类加载器（引导类加载器，Bootstrap ClassLoader）
 - 这个类加载使用C/C++语言实现的，嵌套在JVM内部
 - 它用来加载java的核心库（JAVA_HOME/jre/lib/rt.jar/resources.jar或sun.boot.class.path路径下的内容），用于提供JVM自身需要的类
 - 并不继承自java.lang.ClassLoader,没有父加载器
 - 加载拓展类和应用程序类加载器，并指定为他们的父加载器
 - 处于安全考虑，Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类
2. 拓展类加载器（Extension ClassLoader）
 - java语言编写，由sun.misc.Launcher\$ExtClassLoader实现。
 - 派生于ClassLoader类
 - 父类加载器为启动类加载器
 - 从java.ext.dirs系统属性所指定的目录中加载类库，或从JDK的安装目录的jre/lib/ext子目录（扩展目录）下加载类库。如果用户创建的JAR放在此目录下，也会由拓展类加载器自动加载
3. 应用程序类加载器（系统类加载器，AppClassLoader）
 - java语言编写，由sun.misc.Launcher\$AppClassLoader实现。
 - 派生于ClassLoader类

- 父类加载器为拓展类加载器
- 它负责加载环境变量classpath或系统属性 java.class.path指定路径下的类库
- 该类加载器是程序中默认的类加载器，一般来说，java应用的类都是由它来完成加载
- 通过ClassLoader#getSystemClassLoader()方法可以获取到该类加载器

```
/**
 * 虚拟机自带加载器
 */
public class ClassLoaderTest1 {
    public static void main(String[] args) {
        System.out.println("*****启动类加载器*****");
        URL[] urls = sun.misc.Launcher.getBootstrapClassPath().getURLs();
        //获取BootStrapClassLoader能够加载的api路径
        for (URL e:urls){
            System.out.println(e.toExternalForm());
        }

        //从上面的路径中随意选择一个类 看看他的类加载器是什么
        //Provider位于 /jdk1.8.0_171.jdk/Contents/Home/jre/lib/jsse.jar 下，引导类加载器加载它
        ClassLoader classLoader = Provider.class.getClassLoader();
        System.out.println(classLoader);//null

        System.out.println("*****拓展类加载器*****");
        String extDirs = System.getProperty("java.ext.dirs");
        for (String path : extDirs.split(";")){
            System.out.println(path);
        }

        //从上面的路径中随意选择一个类 看看他的类加载器是什么：拓展类加载器
        ClassLoader classLoader1 = CurveDB.class.getClassLoader();
        System.out.println(classLoader1);//sun.misc.Launcher$ExtClassLoader@4dc63996
    }
}
```

2.3 用户自定义类加载器

为什么

- 隔离加载类
- 修改类加载的方式
- 拓展加载源
- 防止源码泄漏

用户自定义类加载器实现步骤：

1. 开发人员可以通过继承抽象类java.lang.ClassLoader类的方式，实现自己的类加载器，以满足一些特殊的需求
2. 在JDK1.2之前，在自定义类加载器时，总会去继承ClassLoader类并重写loadClass()方法，从而实现自定义的类加载类，但是在JDK1.2之后已不再建议用户去覆盖loadClass()方法，而是建议把自定义的类加载逻辑写在findClass()方法中
3. 在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承URLClassLoader类，这样就可以避免自己去编写findClass()方法及其获取字节码流的方式，使自定义类加载器编写更加简洁。

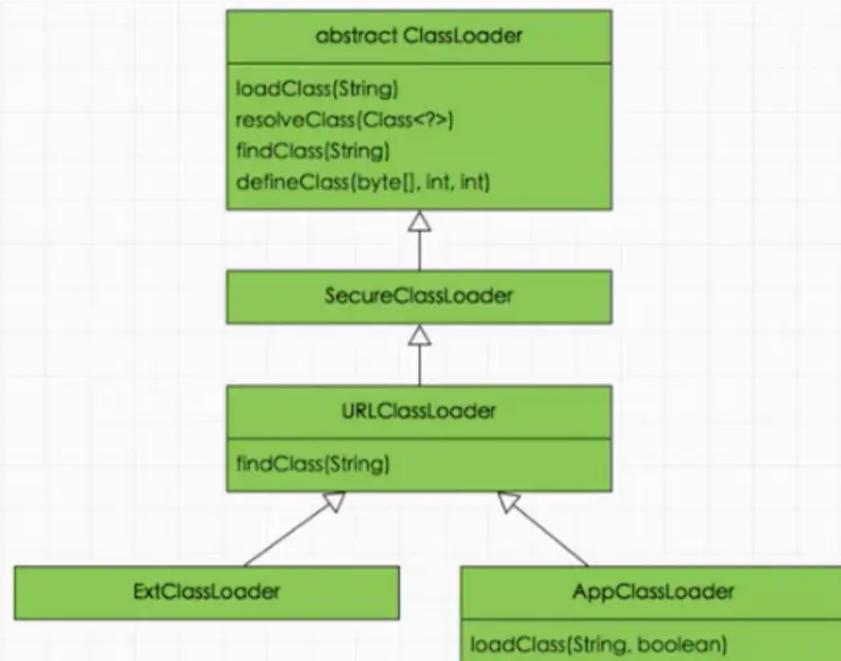
3 ClassLoader的常用方法及获取方法

3.1 ClassLoader类，它是一个抽象类，其后所有的类加载器都继承自**ClassLoader**（不包括启动类加载器）

| 方法名称 | 描述 |
|---|---|
| getParent() | 返回该类加载器的超类加载器 |
| loadClass (String name) | 加载名称为name的类， 返回结果为java.lang.Class类的实例 |
| findClass (String name) | 查找名称为name的类， 返回结果为java.lang.Class类的实例 |
| findLoadedClass (String name) | 查找名称为name的已经被加载过的类， 返回结果为java.lang.Class类的实例 |
| defineClass (String name, byte[] b,int off,int len) | 把字节数组b中的内容转换为一个Java类， 返回结果为java.lang.Class类的实例 |
| defineClass (String name, byte[] b,int off,int len) | 把字节数组b中的内容转换为一个Java类， 返回结果为java.lang.Class类的实例 |

3.2 ClassLoader继承关系

拓展类加载器和系统类加载器间接继承于**ClassLoader**抽象类



`sun.misc.Launcher`
它是一个java虚拟机的
入口应用。

让天下没有难学的技术

3.3 获取**ClassLoader**的途径

获取**ClassLoader**的途径

方式一：获取当前类的ClassLoader****

```
clazz.getClassLoader()
```

方式二：获取当前线程上下文的ClassLoader****

```
Thread.currentThread().getContextClassLoader()
```

方式三：获取系统的ClassLoader****

```
ClassLoader.getSystemClassLoader()
```

方式四：获取调用者的ClassLoader****

```
DriverManager.getCallerClassLoader()
```

```
public class ClassLoaderTest2 {  
    public static void main(String[] args) {  
        try {  
            //1.  
            ClassLoader classLoader = Class.forName("java.lang.String").getClassLoader();  
            System.out.println(classLoader);  
            //2.  
            ClassLoader classLoader1 = Thread.currentThread().getContextClassLoader();  
            System.out.println(classLoader1);  
  
            //3.  
            ClassLoader classLoader2 = ClassLoader.getSystemClassLoader().getParent();  
            System.out.println(classLoader2);  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

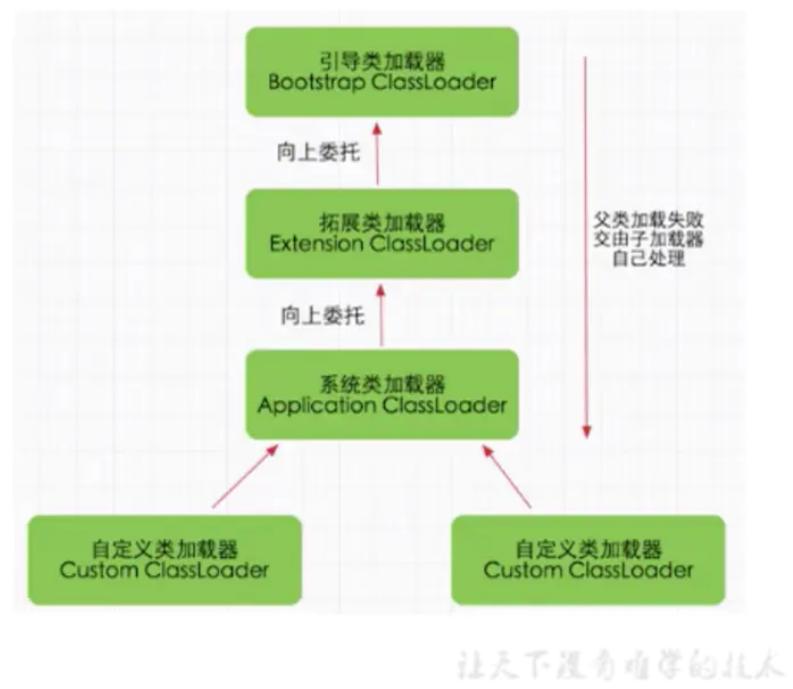
4. 双亲委派机制

Java虚拟机对**class**文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的**class**文件加载到内存生成的**class**对象。而且加载某个类的**class**文件时，**java**虚拟机采用的是双亲委派模式，即把请求交由父类处理，它是一种任务委派模式

4.1 双亲委派机制工作原理

● 工作原理

- 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，请求最终将到达顶层的启动类加载器；
- 3) 如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式。



例

如图，虽然我们自定义了一个java.lang包下的String尝试覆盖核心类库中的String，但是由于双亲委派机制，启动加载器会加载java核心类库的String类（BootStrap启动类加载器只加载包名为java、javax、sun等开头的类），而核心类库中的String并没有main方法

```
ClassLoaderTest
ClassLoaderTest1
ClinitTest1
DeadThreadTest.java
StringTest
StackStruTest
z_JVM_01 简介.md

java.lang
String
jvm.md
JVM_study.iml

External Libraries

Run: String ×
/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin/java ...
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

Process finished with exit code 1
```

```
1 package java.lang;
2
3 public class String {
4     static {
5         System.out.println("我是自定义的String类的静态代码块");
6     }
7
8     public static void main(String[] args) {
9         System.out.println("hello String");
10    }
11
12 }
```

4.2 双亲委派机制的优势

- 避免类的重复加载
- 保护程序安全，防止核心API被随意篡改
 - 自定义类：java.lang.String
 - 自定义类：java.lang.MeDsh（java.lang包需要访问权限，阻止我们用包名自定义类）

File tree:

- ClinitTest1
- DeadThreadTest.java
- StringTest
- StackStruTest
- z_JVM_01 简介.md
- java.lang
 - MeDSH
 - String
- jvm.md

Code editor (MeDSH.java):

```

1 package java.lang;
2
3 public class MeDSH {
4     public static void main(String[] args) {
5         System.out.println("I'm Your Father");
6     }
7 }
8

```

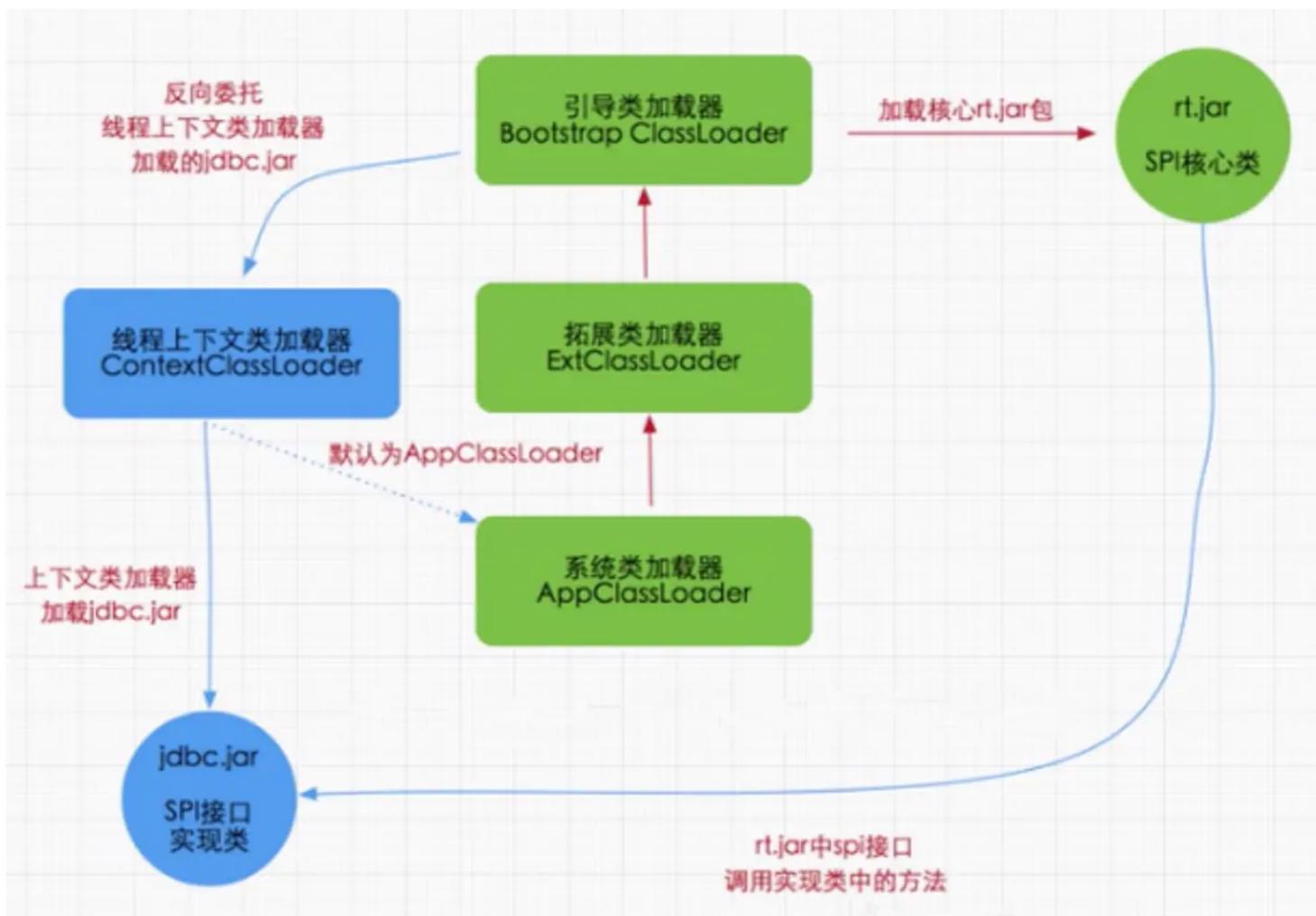
Output window (MeDSH):

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin/java ...
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.SecurityException: Prohibited package name: java.lang
at java.lang.ClassLoader.preDefineClass(ClassLoader.java:662)
at java.lang.ClassLoader.defineClass(ClassLoader.java:761)
at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
at java.net.URLClassLoader$1.run(URLClassLoader.java:362) <1 internal call>
at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349)
at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:495)

```

Process finished with exit code 1



5. 沙箱安全机制

自定义String类，但是在加载子弟敬意String类的时候回率先使用引导类加载器加载，而引导类加载器在加载过程中会先加载jdk自带的文件（rt.jar包中的java\lang\String.class），报错信息说没有main方法就是因为加载的是rt.jar包中的String类。这样可以保证对java核心源代码的保护，这就是沙箱安全机制。

6. 其他

- 在jvm中表示两个class对象是否为同一个类存在的两个必要条件
 - 类的完整类名必须一致，包括包名
 - 加载这个类的ClassLoader（指ClassLoader实例对象）必须相同
- 换句话说，在jvm中，即使这两个类对象（class对象）来源同一个Class文件，被同一个虚拟机所加载，但只要加载它们的ClassLoader实例对象不同，那么这两个类对象也是不相等的。

对类加载器的引用

JVM必须知道一个类型是由启动类加载器加载的还是由用户类加载器加载的。如果一个类型由用户类加载器加载的，那么jvm会将这个类加载器的一个引用作为类型信息的会议部分保存在方法区中。当解析一个类型到另一个类型的引用的时候，JVM需要保证两个类型的加载器是相同的。

类的主动使用和被动使用

java程序对类的使用方式分为：主动使用和被动使用

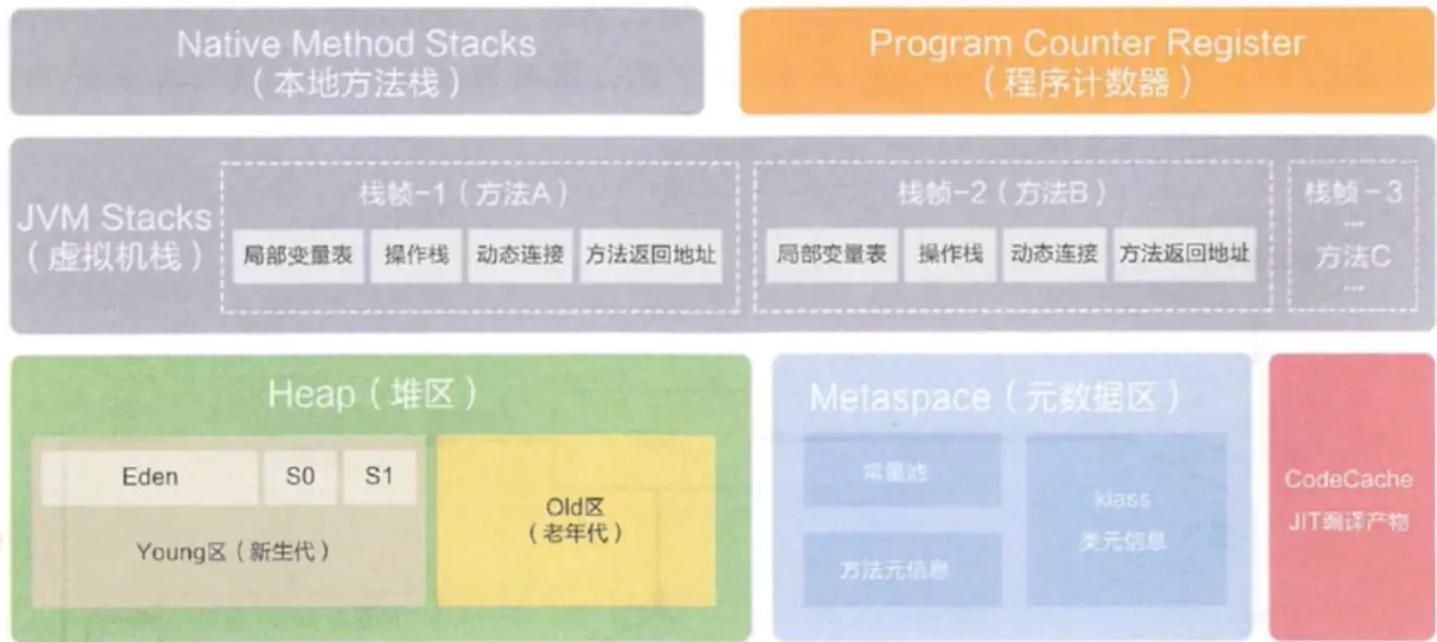
- 主动使用，分为七种情况
 - 创建类的实例
 - 访问某各类或接口的静态变量，或者对静态变量赋值
 - 调用类的静态方法
 - 反射 比如Class.forName([com.dsh.jvm.xxx](#))
 - 初始化一个类的子类
 - java虚拟机启动时被标明为启动类的类
 - JDK 7 开始提供的动态语言支持：
java.lang.invoke.MethodHandle实例的解析结果REF_getStatic、REF_putStatic、
REF_invokeStatic句柄对应的类没有初始化，则初始化
- 除了以上七种情况，其他使用java类的方式都被看作是对类的被动使用，都不会导致类的初始化。

运行时数据区1-[程序计数器+虚拟机栈+本地方法栈]

内存与线程

1 内存

内存是非常重要的系统资源，是硬盘和cpu的中间仓库及桥梁，承载着操作系统和应用程序的实时运行。JVM内存布局规定了JAVA在运行过程中内存申请、分配、管理的策略，保证了JVM的高效稳定运行。不同的jvm对于内存的划分方式和管理机制存在着部分差异（对于Hotspot主要指方法区）



JDK8的元数据区+JIT编译产物 就是JDK8以前的方法区

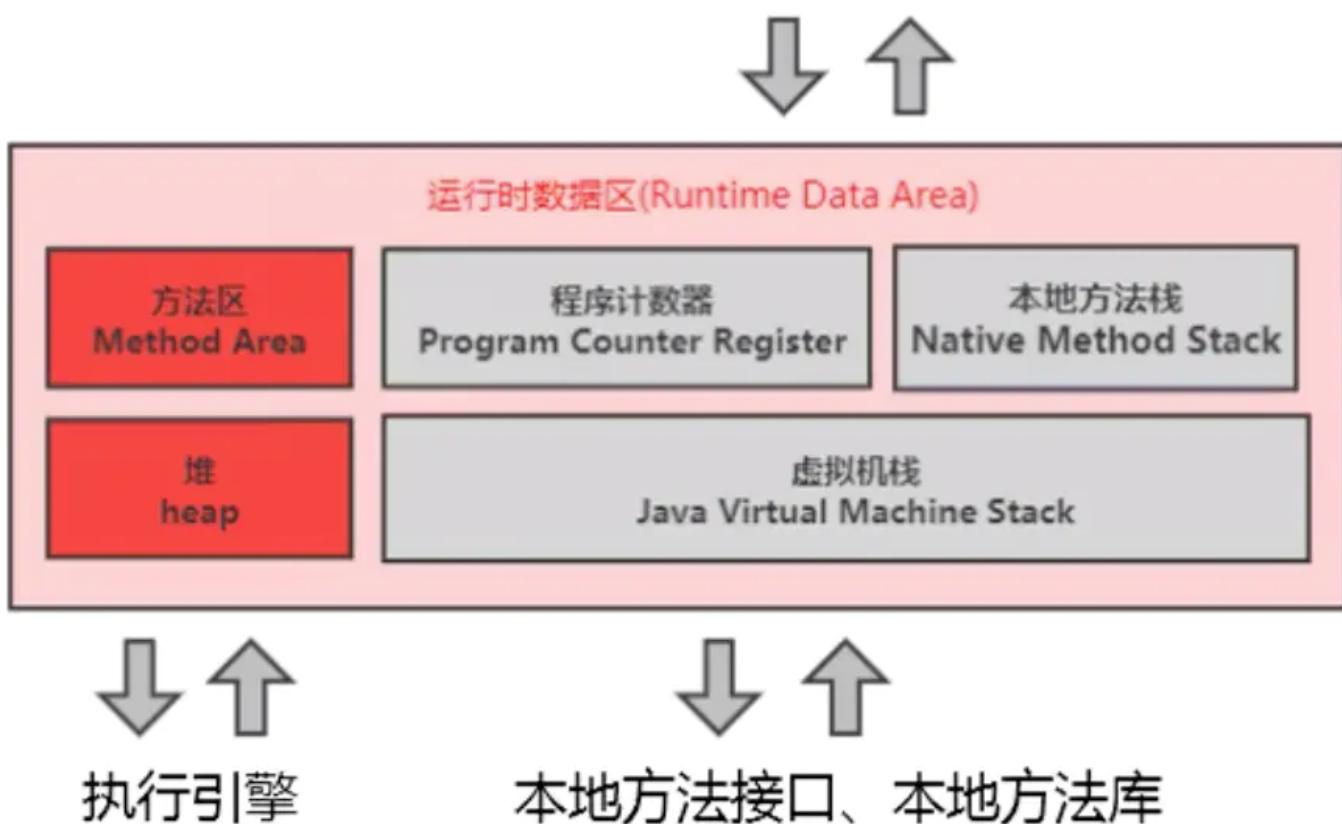
2 分区介绍

java虚拟机定了若干种程序运行期间会使用到的运行时数据区，其中有一些会随着虚拟机启动而创建，随着虚拟机退出而销毁。另外一些则是与线程一一对应的，这些与线程对应的数据区域会随着线程开始和结束而创建和销毁。

如图，灰色的区域为单独线程私有的，红色的为多个线程共享的，即

- 每个线程：独立包括程序计数器、栈、本地栈
- 线程间共享：堆、堆外内存（方法区、永久代或元空间、代码缓存）

类加载器子系统



一般来说，jvm优化95%是优化堆区，5%优化的是方法区

3 线程

- 线程是一个程序里的运行单元，JVM允许一个程序有多个线程并行的执行；
- 在HotSpot JVM，每个线程都与操作系统的本地线程直接映射。
 - 当一个java线程准备好执行以后，此时一个操作系统的本地线程也同时创建。java线程执行终止后。本地线程也会回收。
- 操作系统负责所有线程的安排调度到任何一个可用的CPU上。一旦本地线程初始化成功，它就会调用java线程中的run（）方法。

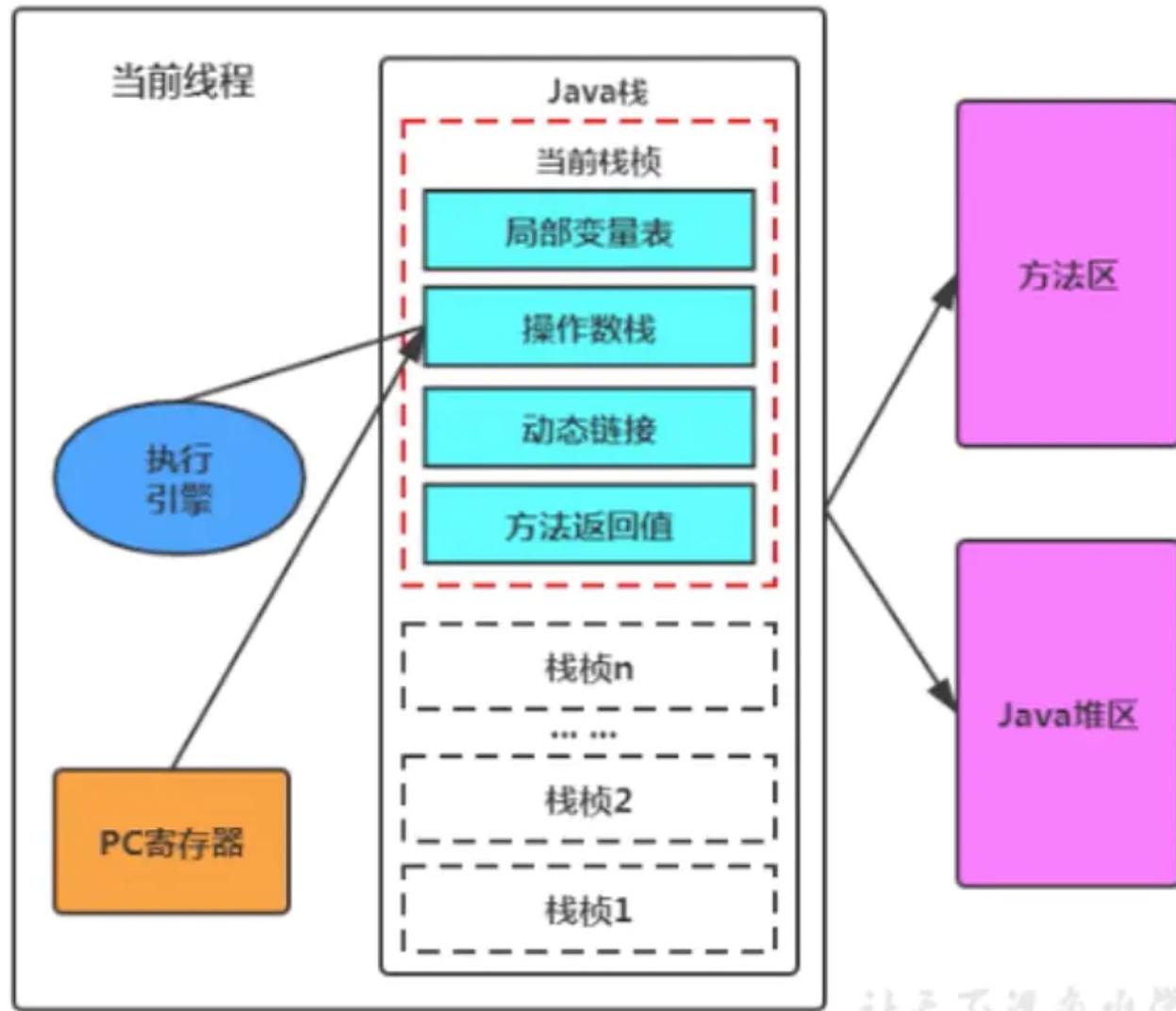
3.1 JVM系统线程

- 如果你使用jconsole或者任何一个调试工具，都能看到在后台有许多线程在运行。这些后台线程不包括调用main方法的main线程以及所有这个main线程自己创建的线程；
- 这些主要的后台系统线程在HotSpot JVM里主要是以下几个：
 - 虚拟机线程这种线程的操作时需要JVM达到安全点才会出现。这些操作必须在不同的线程中发生的原因是他们都需要JVM达到安全点，这样堆才不会变化。这种线程的执行包括“stop-the-world”的垃圾收集，线程栈收集，线程挂起以及偏向锁撤销

- 周期任务线程：这种线程是时间周期事件的提现（比如中断），他们一般用于周期性操作的调度执行。
- GC线程：这种线程对于JVM里不同种类的垃圾收集行为提供了支持
- 编译线程：这种线程在运行时会将字节码编译成本地代码
- 信号调度线程：这种线程接收信号并发送给JVM，在它内部通过调用适当的方法进行处理。

1. 程序计数器（PC寄存器）

JVM中的程序计数器（Program Counter Register）中，Register的命名源于CPU的寄存器，寄存器存储指令相关的现场信息。CPU只有把数据装载到寄存器才能够运行。JVM中的PC寄存器是对屋里PC寄存器的一种抽象模拟



让天下没有难学的技术

1.1 作用

PC寄存器是用来存储指向下一条指令的地址，也即将要执行的指令代码。由执行引擎读取下一条指令。

- 它是一块很小的内存空间，几乎可以忽略不计。也是运行速度最快的存储区域
- 在jvm规范中，每个线程都有它自己的程序计数器，是线程私有的，生命周期与线程的生命周期保持一致
- 任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的java方法的JVM指令地址；或者，如果是在执行native方法，则是未指定值（undefined）。
- 程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成
- 字节码解释器工作时就是通过改变这个计数器的值来选取下一跳需要执行的字节码指令
- 它是唯一一个在java虚拟机规范中没有规定任何OOM情况的区域

1.2 代码示例

利用javap -v xxx.class反编译字节码文件，查看指令等信息

```

public class PCRegister {
    public static void main(String[] args) {
        int i = 10;
        int j = 20;
        int k = i+j;

        String s = "abc";
        System.out.println(i);
        System.out.println(k);
    }
}

descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=5, args_size=1
0: bipush 10
2: istore_1
3: bipush 20
5: istore_2
6: iload_1
7: iload_2
8: iadd
9: istore_3
10: ldc #2          // String abc
12: astore 4
14: getstatic #3   // Field java/lang/System.out:Ljava/io/PrintStream;
17: iload_1
18: invokevirtual #4 // Method java/io/PrintStream.println:(I)V
21: getstatic #3   // Field java/lang/System.out:Ljava/io/PrintStream;
24: iload_3
25: invokevirtual #4 // Method java/io/PrintStream.println:(I)V
28: return

```

1.3 面试常问

1. 使用PC寄存器存储字节码指令地址有什么用呢？ /

为什么使用PC寄存器记录当前线程的执行地址呢？

因为CPU需要不停的切换各个线程，这时候切换回来以后，就得知道接着从哪开始继续执行

JVM的字节码解释器就需要通过改变PC寄存器的值来明确下一条应该执行什么样的字节码指令

2. PC寄存器为什么会设定为线程私有

我们都知道所谓的多线程在一个特定的时间段内指回执行其中某一个线程的方法，CPU会不停滴做任务切换，这样必然会导致经常中断或恢复，如何保证分毫无差呢？为了能够准确地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个线程都分配一个PC寄存器，这样一来各个线程之间便可以进行独立计算，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，一个处理器或者多核处理器中的一个内核，只会执行某个线程中的一条指令。

这样必然导致经常中断或恢复，如何保证分毫无差呢？每个线程在创建后，都会产生自己的程序计数器和栈帧，程序计数器在各个线程之间互不影响。

2. 虚拟机栈

2.1 概述

2.1.1 背景

由于跨平台性的设计，java的指令都是根据栈来设计的。不同平台CPU架构不同，所以不能设计为基于寄存器的。

优点是跨平台，指令集小，编译器容易实现，缺点是性能下降，实现同样的功能需要更多的指令。

2.1.2 内存中的堆与栈

- 栈是运行时的单位，而堆是存储的单位

即：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。

- 一般来讲，对象主要都是放在堆空间的，是运行时数据区比较大的一块
- 栈空间存放基本数据类型的局部变量，以及引用数据类型的对象的引用

2.1.3 虚拟机栈是什么

- java虚拟机栈（Java Virtual Machine Stack），早期也叫Java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（Stack Frame），对应这个一次次的java方法调用。它是线程私有的

- 生命周期和线程是一致的
- 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。
 - 局部变量：相对于成员变量（或属性）
 - 基本数据变量：相对于引用类型变量（类，数组，接口）

2.1.4 栈的特点

- 栈是一种快速有效的分配存储方式，访问速度仅次于PC寄存器（程序计数器）
- JVM直接对java栈的操作只有两个

- 每个方法执行，伴随着进栈（入栈，压栈）
- 执行结束后的出栈工作
- 对于栈来说不存在垃圾回收问题

2.1.5 栈中可能出现的异常

java虚拟机规范允许**Java**栈的大小是动态的或者是固定不变的

- 如果采用固定大小的**Java**虚拟机栈，那每一个线程的**java**虚拟机栈容量可以在线程创建的时候独立选定。如果线程请求分配的栈容量超过**java**虚拟机栈允许的最大容量，**java**虚拟机将会抛出一个 **StackOverflowError** 异常
- 如果**java**虚拟机栈可以动态拓展，并且在尝试拓展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的虚拟机栈，那**java**虚拟机将会抛出一个 **OutOfMemoryError** 异常

2.1.6 设置栈的内存大小

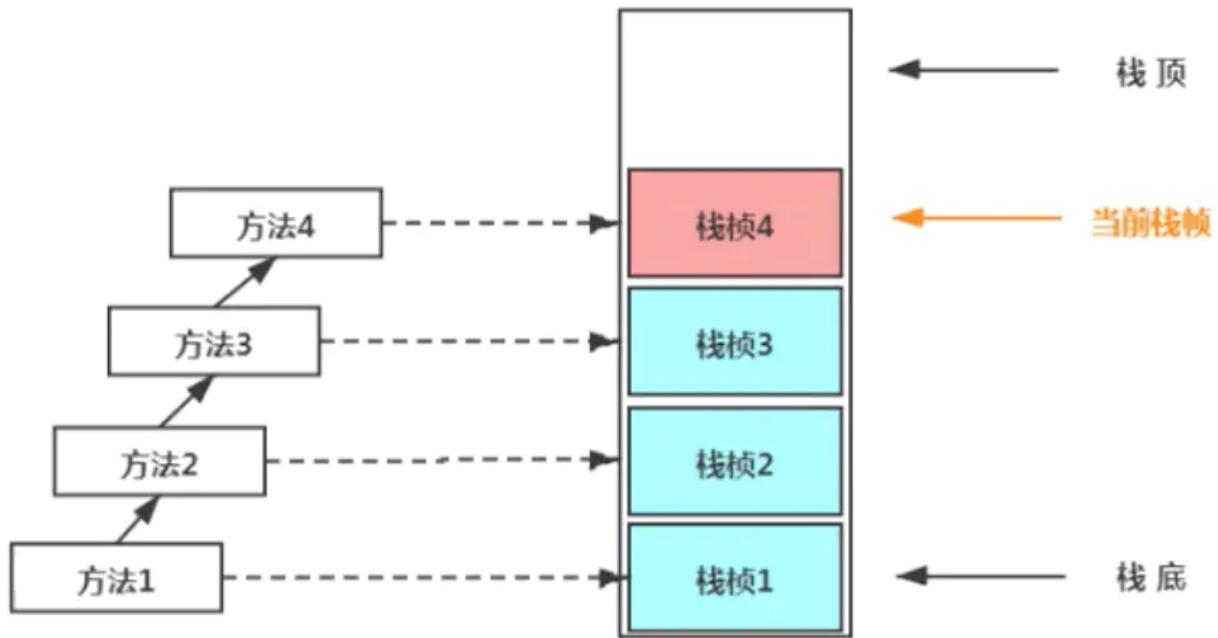
我们可以使用参数-Xss选项来设置线程的最大栈空间，栈的大小直接决定了函数调用的最大可达深度。

(IDEA设置方法：Run>EditConfigurations-VM options 填入指定栈的大小-Xss256k)

2.2 栈的存储结构和运行原理

2.2.1

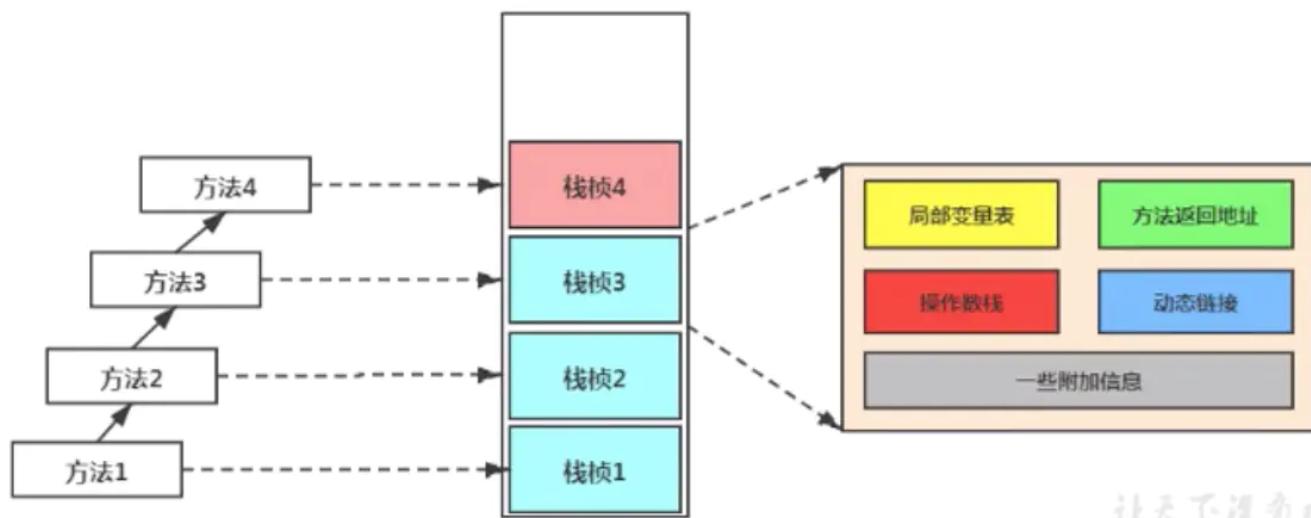
- 每个线程都有自己的栈，栈中的数据都是以栈帧(**Stack Frame**)的格式存在
- 在这个线程上正在执行的每个方法都对应各自的一个栈帧
- 栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息
- JVM直接对**java**栈的操作只有两个，就是对栈帧的压栈和出栈，遵循先进后出/后进先出的原则。
- 在一条活动线程中，一个时间点上，只会有一个活动的栈帧。即只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的，这个栈帧被称为当前栈帧(**Current Frame**)，与当前栈帧对应的方法就是当前方法（**Current Frame**）
- 执行引擎运行的所有字节码指令只针对当前栈帧进行操作
- 如果在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，成为新的当前栈帧。
- 不同线程中所包含的栈帧是不允许相互引用的，即不可能在另一个栈帧中引用另外一个线程的栈帧
- 如果当前方法调用了其他方法，方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧，接着，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧
- **Java**方法有两种返回函数的方式，一种是正常的函数返回，使用**return**指令；另外一种是抛出异常。不管使用哪种方式，都会导致栈帧被弹出。



2.2.2 栈帧的内部结构

每个栈帧中存储着:

- 局部变量表（Local Variables）
- 操作数栈（Operand Stack）(或表达式栈)
- 动态链接（Dynamic Linking）(或执行运行时常量池的方法引用)
- 方法返回地址（Return Address）（或方法正常退出或者异常退出的定义）
- 一些附加信息



2.3 局部变量表（Local Variables）

2.3.1 概述

- 局部变量表也被称为局部变量数组或本地变量表
- 定义为一个数字数组，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（reference），以及returnAddress类型
- 由于局部变量表是建立在线程的栈上，是线程私有的数据，因此不存在数据安全问题
- 局部变量表所需的容量大小是在编译期确定下来的，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
- 方法嵌套调用的次数由栈的大小决定。一般来说，栈越大，方法嵌套调用次数越多。对一个函数而言，他的参数和局部变量越多，使得局部变量表膨胀，它的栈帧就越大，以满足方法调用所需传递的信息增大的需求。进而函数调用就会占用更多的栈空间，导致其嵌套调用次数就会减少。
- 局部变量表中的变量只在当前方法调用中有效。在方法执行时，虚拟机通过使用局部变量表完成参数值到参数变量列表的传递过程。当方法调用结束后，随着方法栈帧的销毁，局部变量表也会随之销毁。

利用javap命令对字节码文件进行解析查看局部变量表，如图：

```

dsh
  jvm
    classloader
    runtimedata
      LocalVariablesTest.class
      PCRegister.class
      StackErrorTest.class
      StackFrameTest.class
      StackTest.class
      StackStruTest.class
      z_JVM_01 简介.md
    java
    META-INF
  src
    com.dsh.jvm
      classloader
Terminal: Local x +
line 11: 15
LocalVariableTable:
  Start  Length  Slot  Name  Signature
    0      16     0   args  [Ljava/lang/String;
    8       8     1   test  Lcom/dsh/jvm/runtimedata/LocalVariablesTest;
   11       5     2   num   I
  
```

也可以在IDEA上安装jclasslib byte viewcoder插件查看字节码信息，以main()方法为例

```

package com.dsh.jvm.runtimedata;

/**
 * 局部变量表
 */
public class LocalVariablesTest {
    public static void main(String[] args) {
        LocalVariablesTest test = new LocalVariablesTest();
        int num = 10;
        test.test1();
    }

    private void test1() {
        int i = 20;
        System.out.println("test1");
    }
}
  
```

Attribute length: 90

Specific info

| Bytecode | Exception table | Misc |
|---|-----------------|------|
| 0 new #2 <com/dsh/jvm/runtimedata/LocalVariables | | |
| 2 dup | | |
| 3 invokespecial #3 <com/dsh/jvm/runtimedata/LocalVariables.<init> | | |
| 4 astore_1 | | |
| 5 bipush 10 | | |
| 6 istore_2 | | |
| 7 aload_1 | | |
| 8 invokespecial #4 <com/dsh/jvm/runtimedata/LocalVariables.test1>() | | |
| 9 return | | |

字节码指令集

Attribute name index: cp_info #12 <LineNumberTable>

Attribute length: 18

Specific info

| Nr. | Start PC | Line Number |
|-----|----------|-------------|
| 0 | 0 | 8 |
| 1 | 8 | 9 |
| 2 | 11 | 10 |
| 3 | 15 | 11 |

代码行号

字节码指令号

General information

Constant Pool

Interfaces

Fields

Methods

<init>

main

Code

LineNumberTable

LocalVariableTable

Attributes

Generic info

Attribute name index: cp_info #13 <LocalVariableTable>

Attribute length: 32

Specific info

| Nr. | Start PC | Length | Index | Name |
|-----|----------|--------|-------|------------------|
| 0 | 0 | 16 | 0 | cp_info #18 args |
| 1 | 8 | 8 | 1 | cp_info #20 test |
| 2 | 11 | 5 | 2 | cp_info #21 num |

2.3.2 变量槽slot的理解与演示

- 参数值的存放总是在局部变量数组的index0开始，到数组长度-1的索引结束
局部变量表，最基本的存储单元是**Slot(变量槽)**
- 局部变量表中存放编译期可知的各种基本数据类型（8种），引用类型（reference），`returnAddress`类型的变量。
- 在局部变量表里，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`）占据两个slot。
 - `byte`、`short`、`char`、`float`在存储前被转换为`int`，`boolean`也被转换为`int`，0表示`false`，非0表示`true`；
 - `long`和`double`则占据两个slot。

| 索引 | 类型 | 参数 |
|----|-----------|----------|
| 0 | int | int k |
| 1 | long | long m |
| 3 | float | float p |
| 4 | double | double q |
| 6 | reference | Object t |

- JVM会为局部变量表中的每一个slot都分配一个访问索引，通过这个索引即可成功访问到局部变量表中指定的局部变量值
- 当一个实例方法被调用的时候，它的方法参数和方法体内部定义的局部变量将会按照顺序被复制到局部变量表中的每一个slot上
- 如果需要访问局部变量表中一个**64bit**的局部变量值时，只需要使用第一个索引即可。（比如：访问long或者double类型变量）
- 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将会存放在**index为0的slot处**，其余的参数按照参数表顺序排列。

2.3.3 slot的重复利用

栈帧中的局部变量表中的槽位是可以重复利用的，如果一个局部变量过了其作用域，那么在其作用域之后申明的新的局部变量就很有可能会复用过期局部变量的槽位，从而达到节省资源的目的。

```
private void test2() {
    int a = 0;
    {
        int b = 0;
        b = a+1;
    }
    //变量c使用之前以及经销毁的变量b占据的slot位置
    int c = a+1;
}
```

2.3.4 静态变量与局部变量的对比及小结

变量的分类：

- 按照数据类型分：
 1. 基本数据类型；
 2. 引用数据类型；
- 按照在类中声明的位置分：
 1. 成员变量：在使用前，都经历过默认初始化赋值
 - static修饰：类变量：类加载linking的准备阶段给类变量默认赋值——>初始化阶段给类变量显式赋值即静态代码块赋值；
 - 不被static修饰：实例变量：随着对象的创建，会在堆空间分配实例变量空间，并进行默认赋值
 2. 局部变量：在使用前，必须要进行显式赋值的！否则，编译不通过 补充：
- 在栈帧中，与性能调优关系最为密切的部分就是局部变量表。在方法执行时，虚拟机使用局部变量表完成方法的传递
- 局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引用的对象都不会被回收

2.4 操作数栈（Operand Stack）

栈：可以使用数组或者链表来实现

- 每一个独立的栈帧中除了包含局部变量表以外，还包含一个后进先出的操作数栈，也可以成为表达式栈
- 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，即入栈（push）或出栈（pop）
 - 某些字节码指令将值压入操作数栈，其余的字节码指令将操作数取出栈，使用他们后再把结果压入栈。（如字节码指令bipush操作）
 - 比如：执行复制、交换、求和等操作

代码举例

```
public void testAddOperation() {  
    byte i = 15;  
    int j = 8;  
    int k = i + j;  
}
```

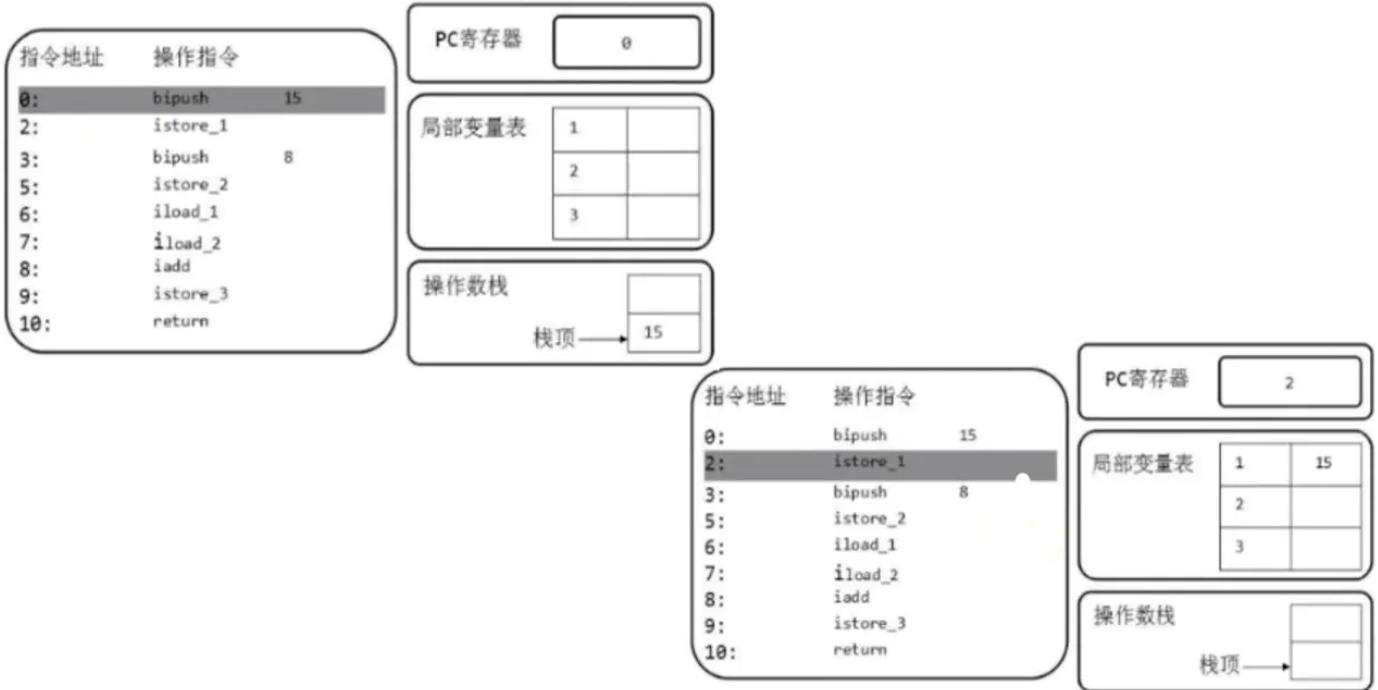
字节码指令信息

```
public void testAddOperation();  
Code:  
0: bipush      15  
2: istore_1  
3: bipush      8  
5: istore_2  
6: iload_1  
7: iload_2  
8: iadd  
9: istore_3  
10: return
```

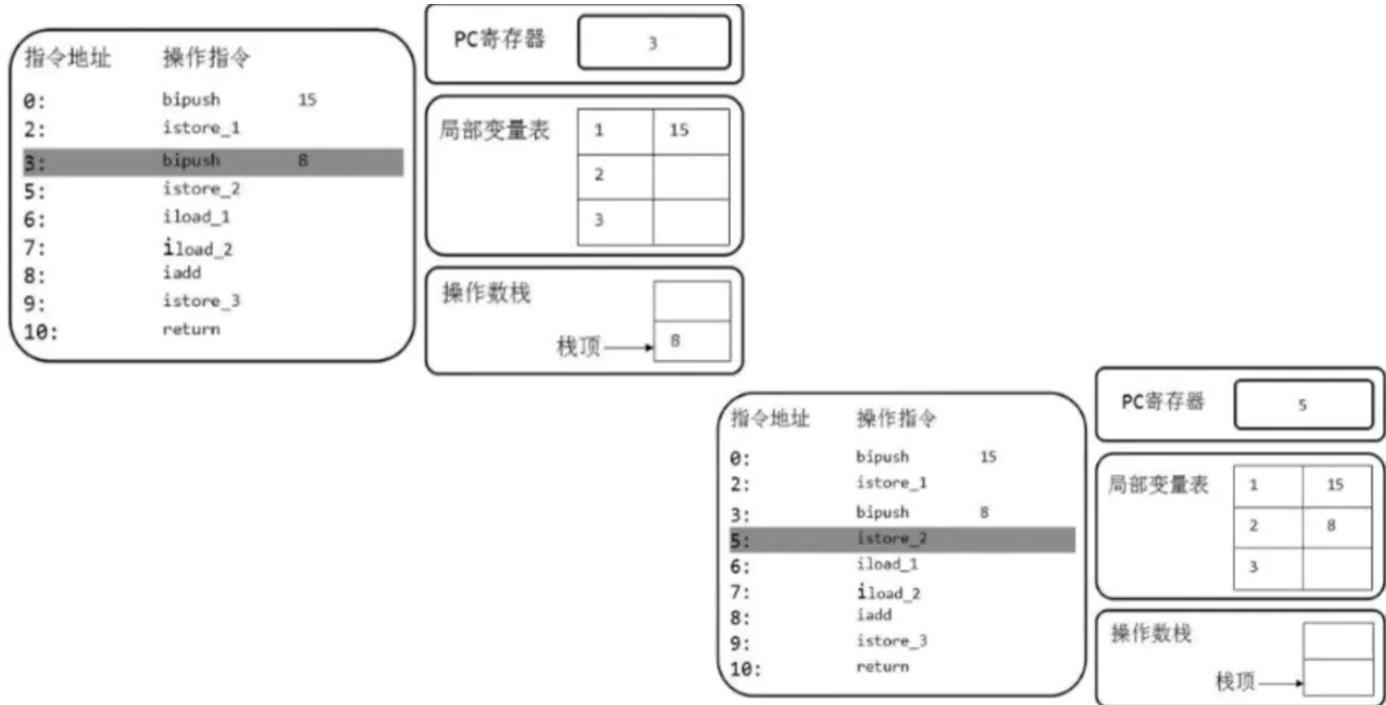
2.4.1 概述

- 操作数栈，主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间。
- 操作数栈就是jvm执行引擎的一个工作区，当一个方法开始执行的时候，一个新的栈帧也会随之被创建出来，这个方法的操作数栈是空的
- 每一个操作数栈都会拥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器就定义好了，保存在方法的code属性中，为max_stack的值。
- 栈中的任何一个元素都是可以任意的java数据类型
 - 32bit的类型占用一个栈单位深度
 - 64bit的类型占用两个栈深度单位
- 操作数栈并非采用访问索引的方式来进行数据访问的，而是只能通过标砖的入栈push和出栈pop操作来完成一次数据访问。
- 如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新PC寄存器中下一条需要执行的字节码指令。
- 操作数栈中的元素的数据类型必须与字节码指令的序列严格匹配，这由编译器在编译期间进行验证，同时在类加载过程中的类验证阶段的数据流分析阶段要再次验证。
- 另外，我们说Java虚拟机的解释引擎是基于栈的执行引擎，其中的栈指的就是操作数栈。
结合上图结合下面的图来看一下一个方法（栈帧）的执行过程

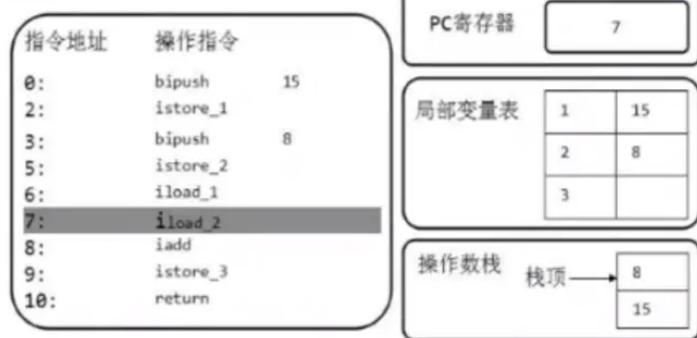
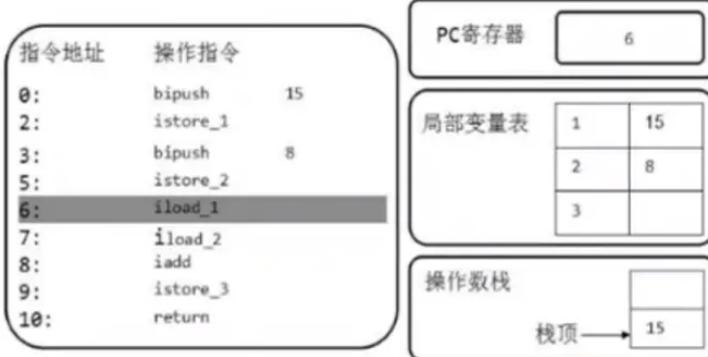
①15入栈；②存储15，15进入局部变量表



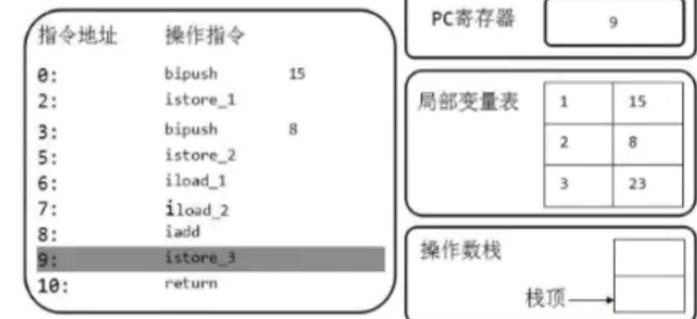
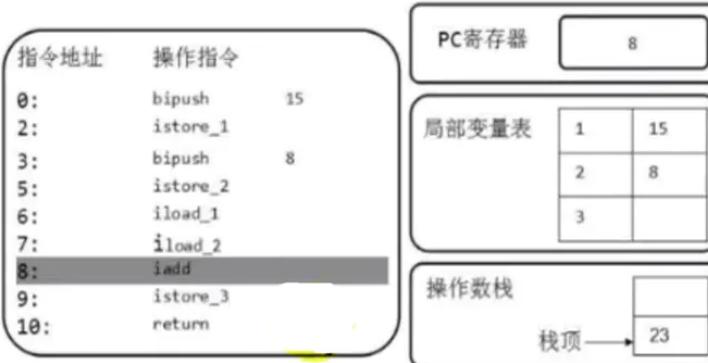
③压入8; ④存储8，8进入局部变量表;



⑤从局部变量表中把索引为1和2的是数据取出来，放到操作数栈；⑥iadd相加操作，8和15出栈



⑦iadd操作结果23入栈；⑧将23存储在局部变量表索引为3的位置上



2.4.2 i++ 和 ++i的区别

```
/*
 * 面试中常被问到的i++和++i的区别
 */
public void add(){
    //第一类问题
    int i1 = 10;
    i1++;

    int i2 = 10;
    ++i2;
}
```

[0] Code

Attributes

i1++和++i2没区别

```

1 0 bipush 10
2 1 istore_1
3 3 iinc 1 by 1
4 6 bipush 10
5 8 istore_2
6 9 iinc 2 by 1
7 12 bipush 10
8 14 istore_3
9 15 iload_3
10 16 iinc 3 by 1
11 19 istore 4
12 21 bipush 10
13 23 istore 5

```

2.4.3 栈顶缓存技术ToS (Top-of-Stack Cashing)

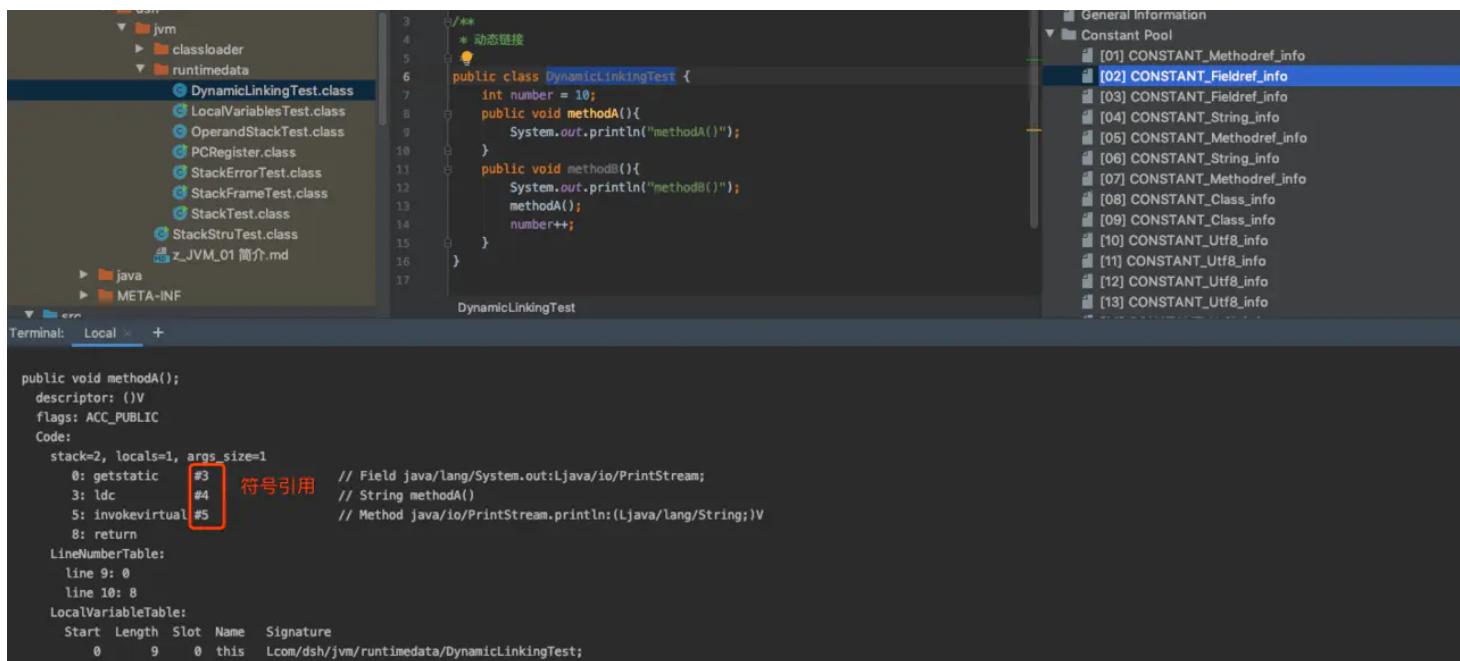
- 基于栈式架构的虚拟机所使用的零地址指令更加紧凑，但完成一项操作的时候必然需要使用更多的入栈和出栈指令，这同时也就意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数
- 由于操作数是存储在内存中的，因此频繁地执行内存读/写操作必然会影响执行速度。为了解决这个问题，HotSpot JVM的设计者们提出了栈顶缓存技术，将栈顶元素全部缓存在物理CPU的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率

2.5 动态链接 (Dynamic Linking)

- 每一个栈帧内部都包含一个指向运行时常量池或该栈帧所属方法的引用。包含这个引用的目的就是为了支持当前方法的代码能够实现动态链接。比如invokedynamic指令
- 在Java源文件被编译成字节码文件中时，所有的变量和方法引用都作为符号引用（symbolic Reference）保存在class文件的常量池里。比如：描述一个方法调用了另外的其他方法时，就是通过常量池中指向方法的符号引用来表示的，那么动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用。

为什么需要常量池呢？

常量池的作用，就是为了提供一些符号和常量，便于指令的识别。



```

Classfile /Users/dongshuhuan/JavaProjects/JVM_study/out/production/JVM_study/com/dsh/jvm/runtimedata/DynamicLinkingTest.class
Last modified 2020-3-16; size 719 bytes
MD5 checksum 94b6a23939d3821cb44265e35cae2ef
Compiled from "DynamicLinkingTest.java"
public class com.dsh.jvm.runtimedata.DynamicLinkingTest
    minor version: 0
    major version: 52
    flags: ACC_PUBLIC, ACC_SUPER
Constant pool: 运行时常量池
#1 = Methodref      #9.#23           // java/lang/Object."<init>":()V
#2 = Fieldref       #8.#24            // com/dsh/jvm/runtimedata/DynamicLinkingTest.number:I
#3 = Fieldref       #25.#26          // java/lang/System.out:Ljava/io/PrintStream;
#4 = String          #27              // methodA()
#5 = Methodref       #28.#29          // java/io/PrintStream.println:(Ljava/lang/String;)V
#6 = String          #30              // methodB()
#7 = Methodref       #8.#31            // com/dsh/jvm/runtimedata/DynamicLinkingTest.methodA:()V
#8 = Class           #32              // com/dsh/jvm/runtimedata/DynamicLinkingTest
#9 = Class           #33              // java/lang/Object
#10 = Utf8           number
#11 = Utf8           I
#12 = Utf8           <init>
#13 = Utf8           ()V

```

2.5.1 方法的调用

在JVM中，将符号引用转换为调用方法的直接引用与方法的绑定机制相关

- 静态链接

当一个字节码文件被装载进JVM内部时，如果被调用的目标方法在编译期可知，且运行期保持不变时。这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接。

- 动态链接

如果被调用的方法在编译期无法被确定下来，也就是说，只能够在程序运行期将调用方法的符号引用转换为直接引用，由于这种引用转换过程具备动态性，因此也就被称之为动态链接。

对应的方法的绑定机制为：早期绑定（Early Binding）和晚期绑定（Late Binding）。绑定是一个字段、方法或者类在符号引用被替换为直接引用的过程，这仅仅发生一次。

- 早期绑定

早期绑定就是指被调用的目标方法如果在编译期可知，且运行期保持不变时，即可将这个方法与所属的类型进行绑定，这样一来，由于明确了被调用的目标方法究竟是哪一个，因此也就可以使用静态链接的方式将符号引用转换为直接引用。

- 晚期绑定

如果被调用的方法在编译期无法被确定下来，只能够在程序运行期根据实际的类型绑定相关的方法，这种绑定方式也就被称之为晚期绑定。

随着高级语言的横空出世，类似于java一样的基于面向对象的编程语言如今越来越多，尽管这类编程语言在语法风格上存在一定的差别，但是它们彼此之间始终保持着一个共性，那就是都支持封装，集成和多态等面向对象特性，既然这一类的编程语言具备多态特性，那么自然也就具备早期绑定和晚期绑定两种绑定方式。

Java中任何一个普通的方法其实都具备虚函数的特征，它们相当于C++语言中的虚函数（C++中则需要使用关键字virtual来显式定义）。如果在Java程序中不希望某个方法拥有虚函数的特征时，则可以使用关键字final来标记这个方法。

2.5.2 虚方法和非虚方法

子类对象的多态性使用前提：①类的继承关系②方法的重写

非虚方法

- 如果方法在编译器就确定了具体的调用版本，这个版本在运行时是不可变的。这样的方法称为非虚方法
- 静态方法、私有方法、**final**方法、实例构造器、父类方法都是非虚方法
- 其他方法称为虚方法

虚拟机中提供了以下几条方法调用指令：

普通调用指令：

1. **invokestatic**: 调用静态方法，解析阶段确定唯一方法版本；
2. **invokespecial**: 调用方法、私有及弗雷方法，解析阶段确定唯一方法版本；
3. **invokevirtual**调用所有虚方法；
4. **invokeinterface**: 调用接口方法；

动态调用指令：

5. **invokedynamic**: 动态解析出需要调用的方法，然后执行。

前四条指令固化在虚拟机内部，方法的调用执行不可人为干预，而**invokedynamic**指令则支持由用户确定方法版本。其中**invokestatic**指令和**invokespecial**指令调用的方法称为非虚方法，其余的（**final**修饰的除外）称为虚方法。

```
/**  
 * 解析调用中非虚方法、虚方法的测试  
 */  
  
class Father {  
    public Father(){  
        System.out.println("Father默认构造器");  
    }  
  
    public static void showStatic(String s){  
        System.out.println("Father show static"+s);  
    }  
  
    public final void showFinal(){  
        System.out.println("Father show final");  
    }  
  
    public void showCommon(){  
        System.out.println("Father show common");  
    }  
  
}  
  
public class Son extends Father{  
    public Son(){  
        super();  
    }  
  
    public Son(int age){  
        this();  
    }  
  
    public static void main(String[] args) {  
        Son son = new Son();  
        son.show();  
    }  
  
    //不是重写的父类方法，因为静态方法不能被重写  
    public static void showStatic(String s){  
        System.out.println("Son show static"+s);  
    }  
  
    private void showPrivate(String s){  
        System.out.println("Son show private"+s);  
    }  
  
    public void show(){  
        //非虚方法如下  
        //invokestatic  
        showStatic(" 大头儿子");  
        //invokestatic  
        super.showStatic(" 大头儿子");  
    }  
}
```

```

//invokespecial
showPrivate(" hello!");
//invokespecial
super.showCommon();
//invokevirtual 因为此方法声明有final 不能被子类重写，所以也认为该方法是非虚方法
showFinal();

//虚方法如下
//invokevirtual
showCommon(); //没有显式加super，被认为是虚方法，因为子类可能重写showCommon
info();

MethodInterface in = null;
//invokeinterface 不确定接口实现类是哪一个 需要重写
in.methodA();

}

public void info(){

}

}

interface MethodInterface {
    void methodA();
}

```

关于invokedynamic指令

- JVM字节码指令集一直比较稳定，一直到jdk7才增加了一个invokedynamic指令，这是Java为了实现【动态类型语言】支持而做的一种改进
- 但是jdk7中并没有提供直接生成invokedynamic指令的方法，需要借助ASM这种底层字节码工具来产生invokedynamic指令。直到Jdk8的Lambda表达式的出现，invokedynamic指令的生成，在java中才有了直接生成方式
- Jdk7中增加的动态语言类型支持的本质是对java虚拟机规范的修改，而不是对java语言规则的修改，这一块相对来讲比较复杂，增加了虚拟机中的方法调用，最直接的受益者就是运行在java平台的动态语言的编译器

动态类型语言和静态类型语言

- 动态类型语言和静态类型语言两者的区别就在于对类型的检查是在编译期还是在运行期，满足前者就是静态类型语言，反之则是动态类型语言。
 - 直白来说 静态语言是判断变量自身的类型信息；动态类型语言是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，这是动态语言的一个重要特征
- Java是静态类型语言（尽管lambda表达式为其增加了动态特性），js, python是动态类型语言。

2.5.3 方法重写的本质

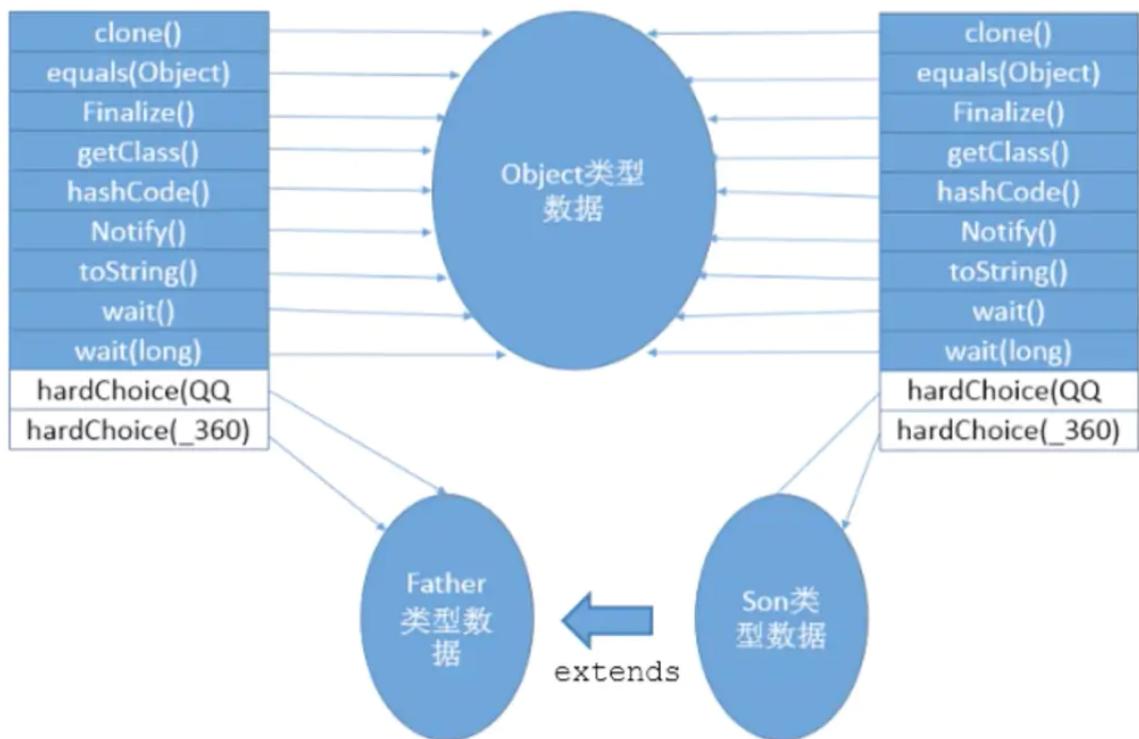
1. 找到操作数栈的第一个元素所执行的对象的实际类型，记作C。
2. 如果在类型C中找到与常量中的描述符合简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过，则返回 `java.lang.IllegalAccessError` 异常。
3. 否则，按照继承关系从下往上依次对c的各个父类进行第二步的搜索和验证过程。
4. 如果始终没有找到合适的方法，则抛出 `java.lang.AbstractMethodError` 异常。

`IllegalAccessException` 介绍 程序视图访问或修改一个属性或调用一个方法，这个属性或方法，你没有权限访问。一般的，这个会引起编译器异常。这个错误如果发生在运行时，就说明一个类发生了不兼容的改变。

2.5.4 虚方法表

- 在面向对象编程中，会很频繁地使用到动态分派，如果在每次动态分派的过程中都要重新在累的方法元数据中搜索合适的目标的话就可能影响到执行效率。因此，为了提高性能，jvm采用在类的方法区建立一个虚方法表（virtual method table）（非虚方法不会出现在表中）来实现。使用索引表来代替查找。
- 每个类中都有一个虚方法表，表中存放着各个方法的实际入口。
- 那么虚方法表什么时候被创建？虚方法表会在类加载的链接阶段被创建并开始初始化，类的变量初始值准备完成之后，jvm会把该类的虚方法表也初始化完毕。

举例1：



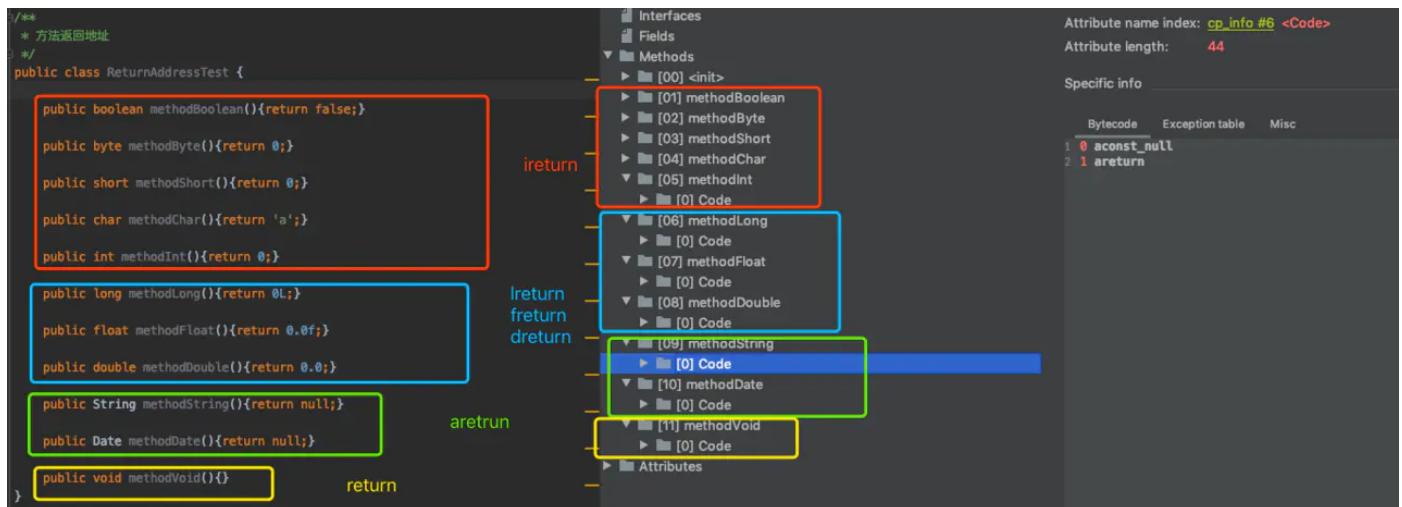
2.6 方法返回地址 (Return Address)

- 存放调用该方法的PC寄存器的值。

- 一个方法的结束，有两种方式：
 - 正常执行完成
 - 出现未处理的异常，非正常退出
- 无论通过哪种方式退出，在方法退出后都返回到该方法被调用的位置。方法正常退出时，调用者的pc计数器的值作为返回地址，即调用该方法的指令的下一条指令的地址。而通过异常退出时，返回地址是要通过异常表来确定，栈帧中一般不会保存这部分信息。
- 本质上，方法的退出就是当前栈帧出栈的过程。此时，需要恢复上层方法的局部变量表、操作数栈、将返回值也如调用者栈帧的操作数栈、设置PC寄存器值等，让调用者方法继续执行下去。
- 正常完成出口和异常完成出口的区别在于：通过异常完成出口退出的不会给他的上层调用者产生任何的返回值。

当一个方法开始执行后，只要两种方式可以退出这个方法：

1. 执行引擎遇到任意一个方法返回的字节码指令（return），会有返回值传递给上层的方法调用者，简称正常完成出口：
 - 一个方法在正常调用完成之后究竟需要使用哪一个返回指令还需要根据方法返回值的实际数据类型而定
 - 在字节码指令中，返回指令包含ireturn（当返回值是boolean、byte、char、short和int类型时使用）、lreturn、freturn、dreturn以及areturn，另外还有一个return指令供声明为void的方法、实例初始化方法、类和接口的初始化方法使用
2. 在方法执行的过程中遇到了异常（Exception），并且这个异常没有在方法内进行处理，也就是只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，简称异常完成出口
方法执行过程中抛出异常时的异常处理，存储在一个异常处理表，方便在发生异常的时候找到处理异常的代码。



2.7 虚拟机栈的5道面试题

1. 举例栈溢出的情况？（StackOverflowError）
 - 递归调用等，通过-Xss设置栈的大小；
2. 调整栈的大小，就能保证不出现溢出么？
 - 不能 如递归无限次数肯定会溢出，调整栈大小只能保证溢出的时间晚一些

3. 分配的栈内存越大越好么?
 - 不是 会挤占其他线程的空间
4. 垃圾回收是否会涉及到虚拟机栈?

| 内存块 | error | GC |
|---------|-------|----|
| 程序计数器 | × | × |
| 本地方法栈 | ✓ | × |
| jvm虚拟机栈 | ✓ | × |
| 堆 | ✓ | ✓ |
| 方法区 | ✓ | ✓ |

5. 方法中定义的局部变量是否线程安全?

```
/**  
 * 面试题：  
 * 方法中定义的局部变量是否线程安全？具体情况具体分析  
 *  
 * 何为线程安全？  
 *     如果只有一个线程可以操作此数据，则毙是线程安全的。  
 *     如果有多个线程操作此数据，则此数据是共享数据。如果不考虑同步机制的话，会存在线程安全问题  
 *  
 * StringBuffer是线程安全的， StringBuilder不是  
 */  
public class StringBuilderTest {  
  
    //s1的声明方式是线程安全的  
    public static void method1(){  
        StringBuilder s1 = new StringBuilder();  
        s1.append("a");  
        s1.append("b");  
    }  
  
    //stringBuilder的操作过程：是不安全的，因为method2可以被多个线程调用  
    public static void method2(StringBuilder stringBuilder){  
        stringBuilder.append("a");  
        stringBuilder.append("b");  
    }  
  
    //s1的操作：是线程不安全的 有返回值，可能被其他线程共享  
    public static StringBuilder method3(){  
        StringBuilder s1 = new StringBuilder();  
        s1.append("a");  
        s1.append("b");  
        return s1;  
    }  
  
    //s1的操作：是线程安全的， StringBuilder的toString方法是创建了一个新的String， s1在内部消亡了  
    public static String method4(){  
        StringBuilder s1 = new StringBuilder();  
        s1.append("a");  
        s1.append("b");  
        return s1.toString();  
    }  
  
    public static void main(String[] args) {  
        StringBuilder s = new StringBuilder();  
        new Thread(()->{  
            s.append("a");  
            s.append("b");  
        }).start();  
  
        method2(s);  
    }  
}
```

}

3.本地方法栈

- 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
- 本地方法栈，也是线程私有的。
- 允许被实现成固定或者是可动态拓展的内存大小。（在内存溢出方面是相同的）
 - 如果线程请求分配的栈容量超过本地方法栈允许的最大容量，Java虚拟机将会抛出一个**StackOverflowError**异常。
 - 如果本地方法栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的本地方法栈，那么java虚拟机将会抛出一个**OutOfMemoryError**异常。
- 本地方法是使用C语言实现的
- 它的具体做法是Native Method Stack中登记native方法，在Execution Engine执行时加载本地方法库。
- 当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。它和虚拟机拥有同样的权限
 - 本地方法可以通过本地方法接口来访问虚拟机内部的运行时数据区
 - 它甚至可以直接使用本地处理器中的寄存器
 - 直接从本地内存的堆中分配任意数量的内存
- 并不是所有的JVM都支持本地方法。因为Java虚拟机规范并没有明确要求本地方法栈的使用语言、具体实现方式、数据结构等。如果JVM产品不打算支持native方法，也可以无需实现本地方法栈。
- 在hotSpot JVM中，直接将本地方法栈和虚拟机栈合二为一。

为什么要使用Native Method？

java使用起来非常方便，然而有些层次的任务用java实现起来不容易，或者我们对程序的效率很在意时，问题就来了。

- 与java环境外交互：

有时java应用需要与java外面的环境交互，这是本地方法存在的主要原因。

你可以想想java需要与一些底层系统，如操作系统或某些硬件交换信息时的情况。本地方法正式这样的一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解java应用之外的繁琐细节。

- 与操作系统交互

JVM支持着java语言本身和运行库，它是java程序赖以生存的平台，它由一个解释器（解释字节码）和一些连接到本地代码的库组成。然而不管怎样，它毕竟不是一个完整的系统，它经常依赖于一些底层系统的支持。这些底层系统常常是强大的操作系统。通过使用本地方法，我们得以用java实现了jre的与底层系统的交互，甚至jvm的一些部分就是用C写的。还有，如果我们要使用一些java语言本身没有提供封装的操作系统特性时，我们也需要使用本地方法。

- Sun's Java

Sun的解释器是用C实现的，这使得它能像一些普通的C一样与外部交互。jre大部分是用java实现的，它也通过一些本地方法与外界交互。例如：类 `java.lang.Thread` 的 `setPriority()` 方法是用Java实现的，但是它实现调用的事该类里的本地方法 `setPriority0()`。这个本地方法是用C实现的，并被植入JVM内部，在 Windows 95 的平台上，这个本地方法最终将调用 Win32 SetPriority() API。这是一个本地方法的具体实现由JVM直接提供，更多的情况是本地方法由外部的动态链接库（`external dynamic link library`）提供，然后被JVM调用。

运行时数据区2-堆

1.核心概述

一个进程对应一个jvm实例，一个运行时数据区，又包含多个线程，这些线程共享了方法区和堆，每个线程包含了程序计数器、本地方法栈和虚拟机栈。

1. 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
2. Java堆区在JVM启动的时候即被创建，其空间大小也就确定了。是JVM管理的最大一块内存空间（堆内存的大小是可以调节的）
3. 《Java虚拟机规范》规定，堆可以处于物理上不连续的内存空间中，但在逻辑上它应该被视为连续的
4. 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（`TLAB:Thread Local Allocation Buffer`）。（面试问题：堆空间一定是所有线程共享的么？不是，`TLAB`线程在堆中独有的）
5. 《Java虚拟机规范》中对java堆的描述是：所有的对象实例以及数组都应当在运行时分配在堆上。
 - 从实际使用的角度看，“几乎”所有的对象的实例都在这里分配内存（‘几乎’是因为可能存储在栈上）
6. 数组或对象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在堆中的位置
7. 在方法结束后，堆中的对象不会马上被移除，仅仅在垃圾收集的时候才会被移除
8. 堆，是GC(Garbage Collection，垃圾收集器)执行垃圾回收的重点区域

1.1 配置jvm及查看jvm进程

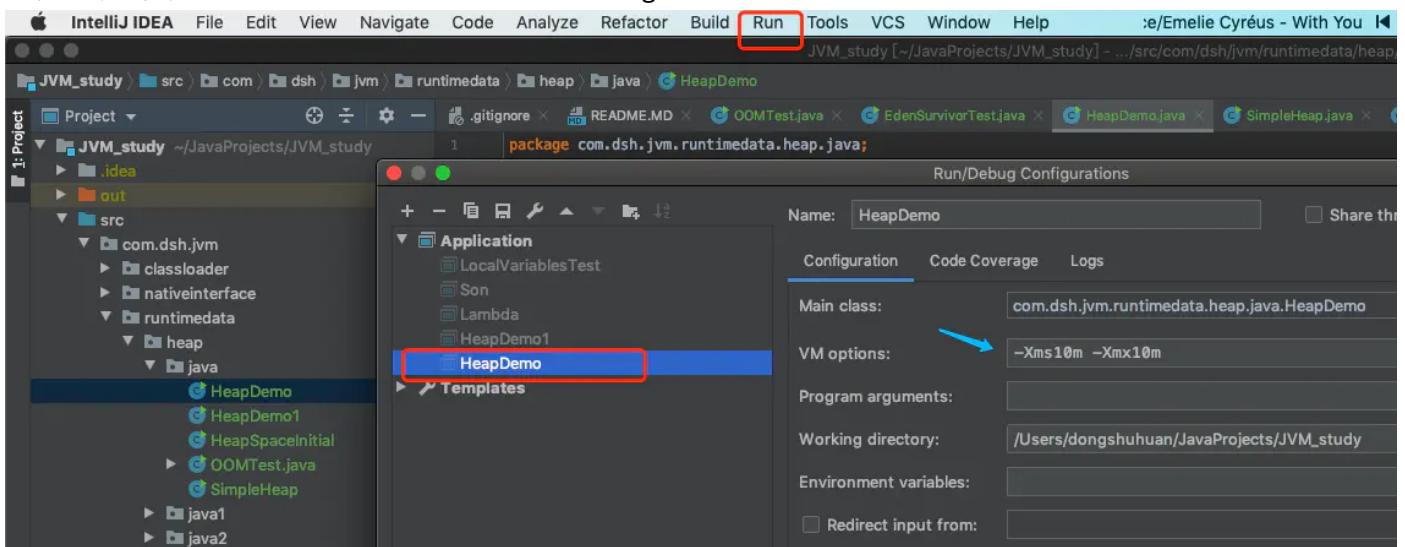
- 编写HeapDemo/HeapDemo1代码

```

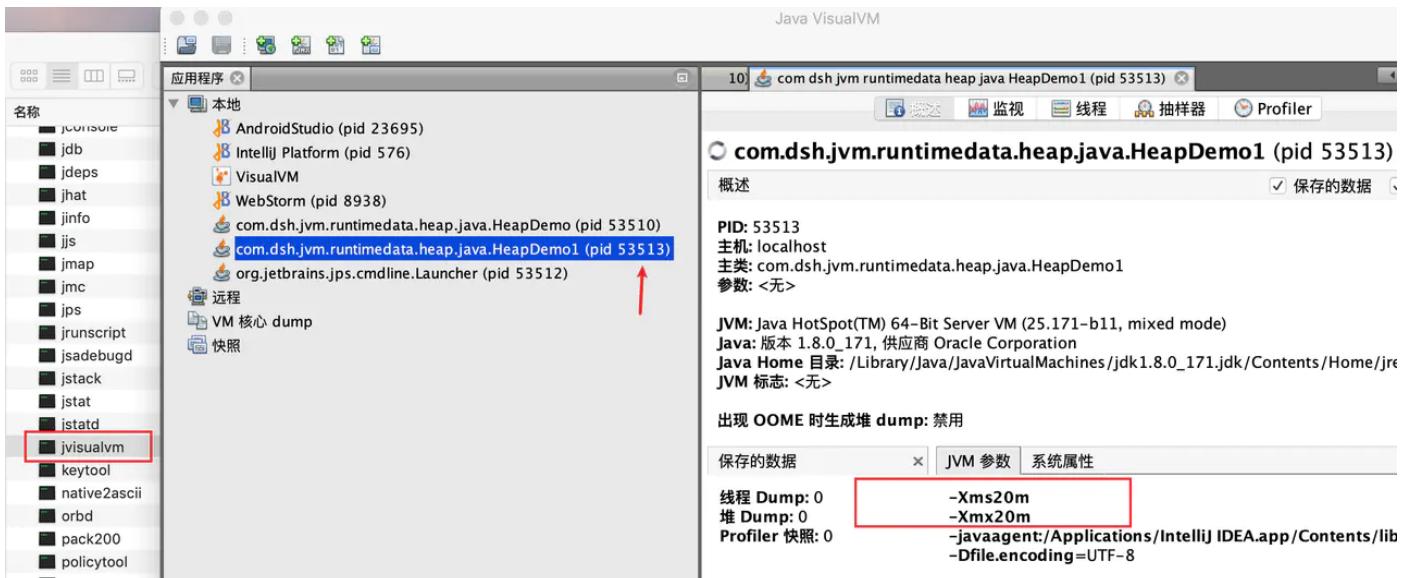
public class HeapDemo {
    public static void main(String[] args) {
        System.out.println("start...");
        try {
            Thread.sleep(1000000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("end...");
    }
}

```

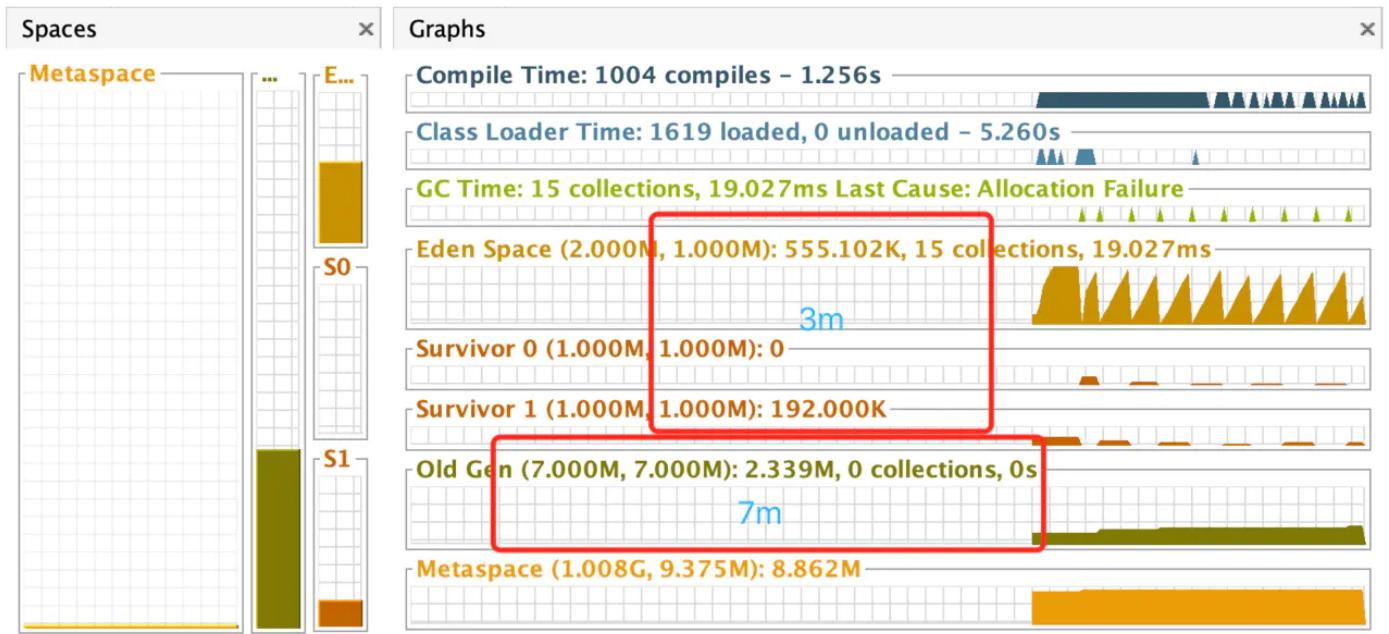
- 首先对虚拟机进行配置，如图 Run>Edit configurations



- 在jdk目录，我的是 /Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin 下找到 jvisualvm 运行（或者直接终端运行），查看进程，可以看到我们设置的配置信息



- 可以看到HeapDemo配置-Xms10m，分配的10m被分配给了新生代3m和老年代7m



1.2 分析SimpleHeap的jvm情况

```

public class SimpleHeap {
    private int id;//属性、成员变量

    public SimpleHeap(int id) {
        this.id = id;
    }

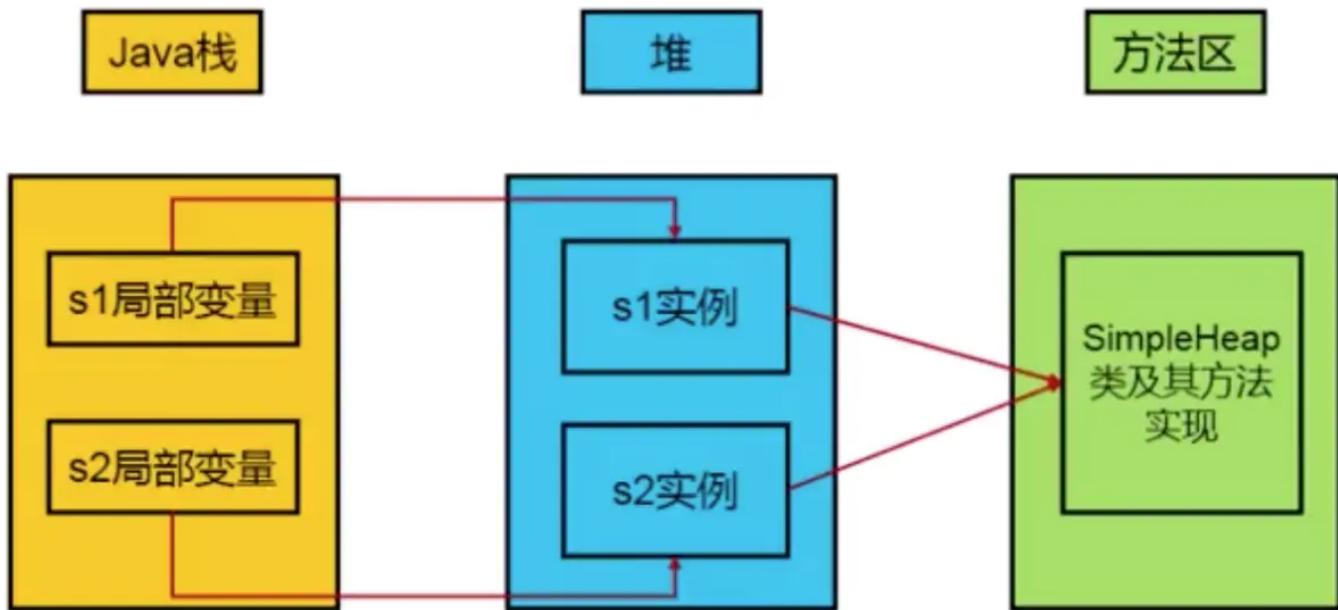
    public void show() {
        System.out.println("My ID is " + id);
    }

    public static void main(String[] args) {
        SimpleHeap s1 = new SimpleHeap(1);
        SimpleHeap s2 = new SimpleHeap(2);

        int[] arr = new int[10];

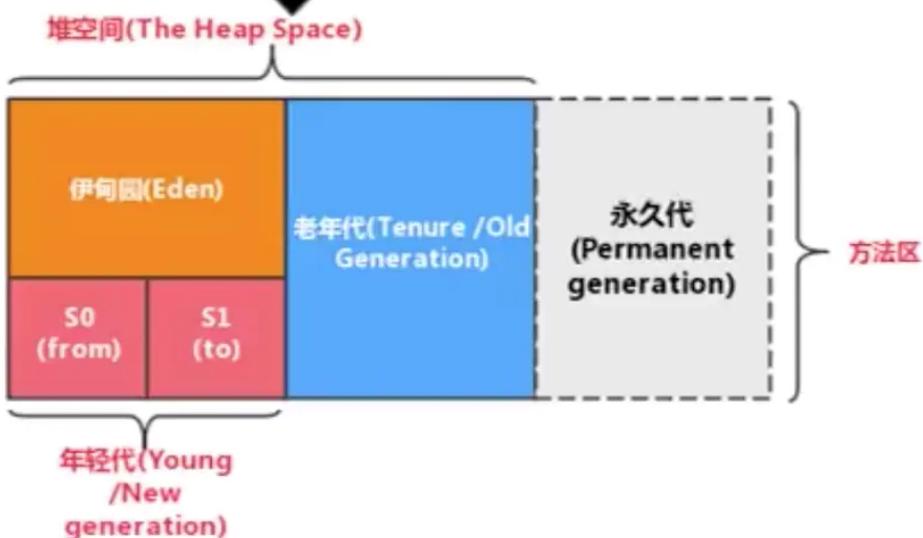
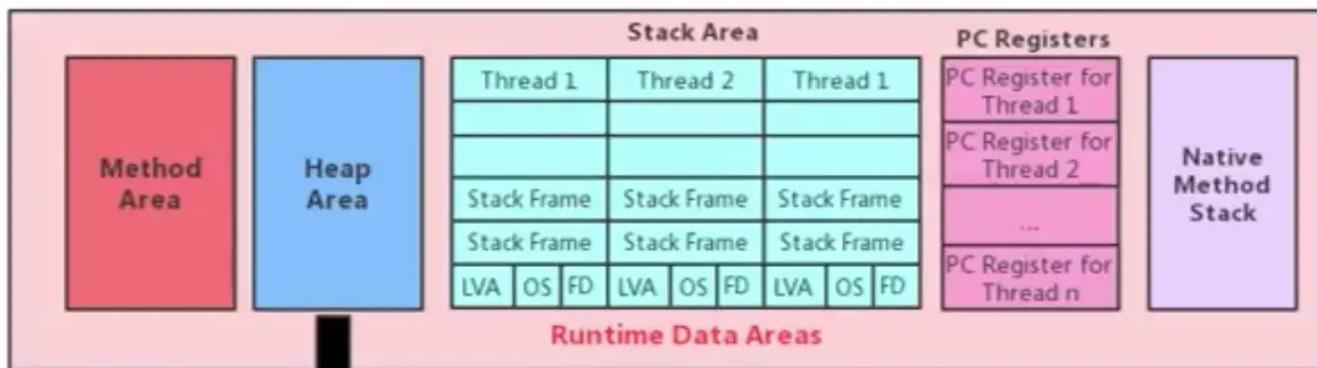
        Object[] arr1 = new Object[10];
    }
}

```

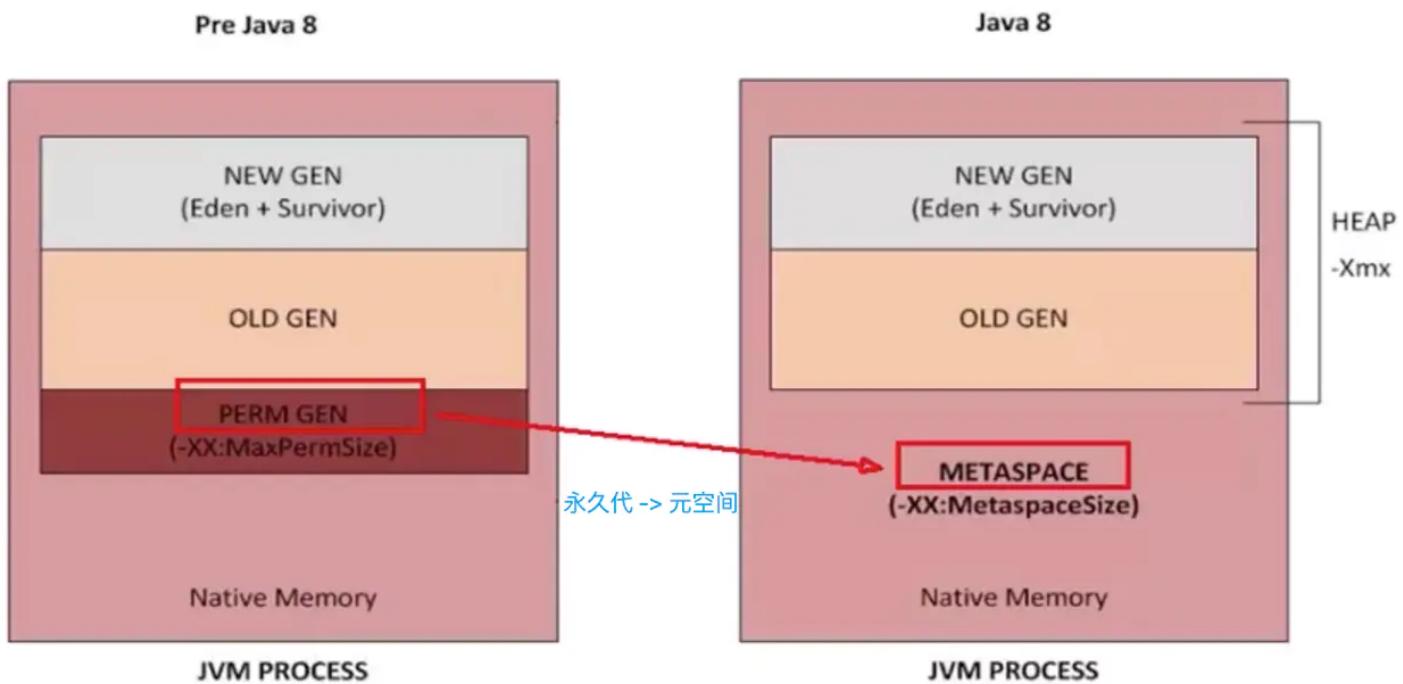


1.3 堆的细分内存结构

- JDK 7以前: 新生区+养老区+永久区
 - Young Generation Space : 又被分为 Eden 区和 Survivor 区 Young/New
 - Tenure generation Space : Old/Tenure
 - Permanent Space : Perm



- JDK 8以后: 新生区+养老区+元空间
 - Young Generation Space : 又被分为 Eden 区和 Survivor 区 Young/New
 - Tenure generation Space : Old/Tenure
 - Meta Space : Meta



2.设置堆内存大小与OOM

- Java堆区用于存储java对象实例，堆的大小在jvm启动时就已经设定好了，可以通过 `-Xmx` 和 `-Xms` 来进行设置
 - `-Xms` 用于表示堆的起始内存，等价于 `-XX:InitialHeapSize`
 - `-Xms` 用来设置堆空间（年轻代+老年代）的初始内存大小
 - `-X` 是jvm的运行参数
 - `ms` 是memory start
 - `-Xmx` 用于设置堆的最大内存，等价于 `-XX:MaxHeapSize`
- 一旦堆区中的内存大小超过 `-Xmx` 所指定的最大内存时，将会抛出OOM异常
- 通常会将 `-Xms` 和 `-Xmx` 两个参数配置相同的值，其目的就是为了能够在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
- 默认情况下，初始内存大小：物理内存大小/64;最大内存大小：物理内存大小/4
 - 手动设置：`-Xms600m -Xmx600m`
- 查看设置的参数：
 - 方式一：终端输入 `jps`，然后 `jstat -gc` 进程id
 - 方式二：（控制台打印）Edit Configurations->VM Options 添加 `-XX:+PrintGCDetails`

2.1 查看堆内存大小

```
public class HeapSpaceInitial {  
    public static void main(String[] args) {  
  
        //返回Java虚拟机中的堆内存总量  
        long initialMemory = Runtime.getRuntime().totalMemory() / 1024 / 1024;  
        //返回Java虚拟机试图使用的最大堆内存量  
        long maxMemory = Runtime.getRuntime().maxMemory() / 1024 / 1024;  
  
        System.out.println("-Xms : " + initialMemory + "M"); // -Xms : 245M  
        System.out.println("-Xmx : " + maxMemory + "M"); // -Xmx : 3641M  
  
        System.out.println("系统内存大小为: " + initialMemory * 64.0 / 1024 + "G"); // 系统内存大小为  
        System.out.println("系统内存大小为: " + maxMemory * 4.0 / 1024 + "G"); // 系统内存大小为: 14.  
  
        try {  
            Thread.sleep(1000000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

2.2 堆大小分析

设置堆大小为600m，打印出的结果为575m，这是因为幸存者区S0和S1各占据了25m，但是他们始终有一个是空的，存放对象的是伊甸园区和一个幸存者区

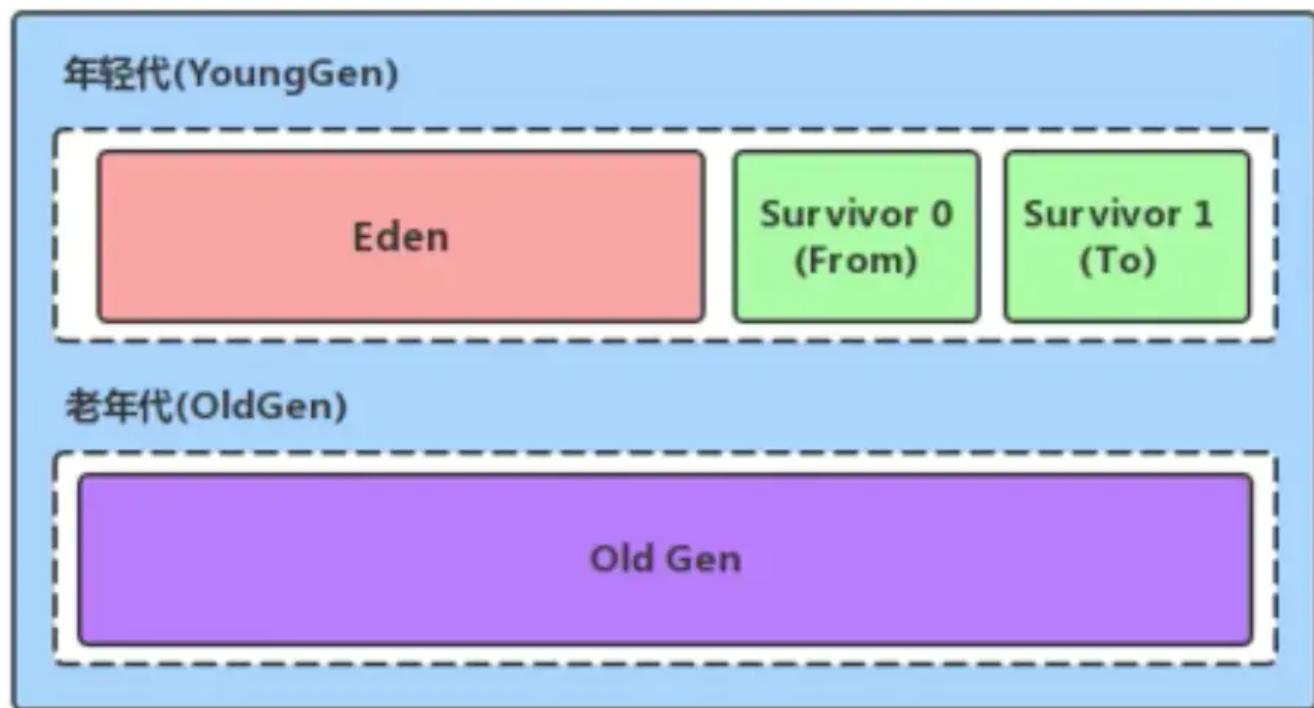
```
dongshuhuan@MacBook-Pro:~$ jps  
39248  PCRegister  
53186 Launcher  
53187 HeapSpaceInitial  
53189 Jps  
47878 StackFrameTest  
23695 StackErrorTest  
dongshuhuan@MacBook-Pro:~$ jstat -gc 53187  


| SOC     | SIC     | SOU | SIU | EC       | EU      | OC       |
|---------|---------|-----|-----|----------|---------|----------|
| 25600.0 | 25600.0 | 0.0 | 0.0 | 153600.0 | 15360.5 | 409600.0 |

  
600-25600/1024 = 575  
S0 或 S1 有一个区是空的  
HeapSpaceInitial main()  
-Xms : 575M  
-Xmx : 575M
```

3. 年轻代与老年代

- 存储在JVM中的java对象可以被划分为两类：
 - 一类是生命周期较短的瞬时对象，这类对象的创建和消亡都非常迅速
 - 另外一类对象时生命周期非常长，在某些情况下还能与JVM的生命周期保持一致
 - Java堆区进一步细分可以分为年轻代（YoungGen）和老年代（OldGen）
 - 其中年轻代可以分为Eden空间、Survivor0空间和Survivor1空间（有时也叫from区，to区）



- 配置新生代与老年年代在堆结构的占比
 - 默认 `-XX: NewRatio=2`，表示新生代占1，老年年代占2，新生代占整个堆的1/3

- 可以修改 `-XX:NewRatio=4`，表示新生代占1，老年代占4，新生代占整个堆的1/5
- 在hotSpot中，`Eden` 空间和另外两个 `Survivor` 空间缺省所占的比例是8: 1: 1 (测试的时候是6: 1: 1)，开发人员可以通过选项 `-XX:SurvivorRatio` 调整空间比例，如 `-XX:SurvivorRatio=8`
- 几乎所有的Java对象都是在`Eden`区被new出来的
- 绝大部分的Java对象都销毁在新生代了 (IBM公司的专门研究表明，新生代80%的对象都是“朝生夕死”的)
- 可以使用选项 `-Xmn` 设置新生代最大内存大小 (这个参数一般使用默认值就好了)

测试代码

```
/*
 * -Xms600m -Xmx600m
 *
 * -XX:NewRatio : 设置新生代与老年代的比例。默认值是2.
 * -XX:SurvivorRatio : 设置新生代中Eden区与Survivor区的比例。默认值是8
 * -XX:-UseAdaptiveSizePolicy : 关闭自适应的内存分配策略 '-'关闭,'+'打开 (暂时用不到)
 * -Xmn:设置新生代的空间的大小。 (一般不设置)
 *
 */
public class EdenSurvivorTest {
    public static void main(String[] args) {
        System.out.println("我只是来打个酱油~");
        try {
            Thread.sleep(1000000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

4.图解对象分配过程

4.1

为新对象分配内存是一件非常严谨和复杂的任务，JVM的设计者们不仅需要考虑内存如何分配、在哪里分配的问题，并且由于内存分配算法与内存回收算法密切相关，所以还需要考虑GC执行完内存回收后是否会在内存空间中产生内存碎片。

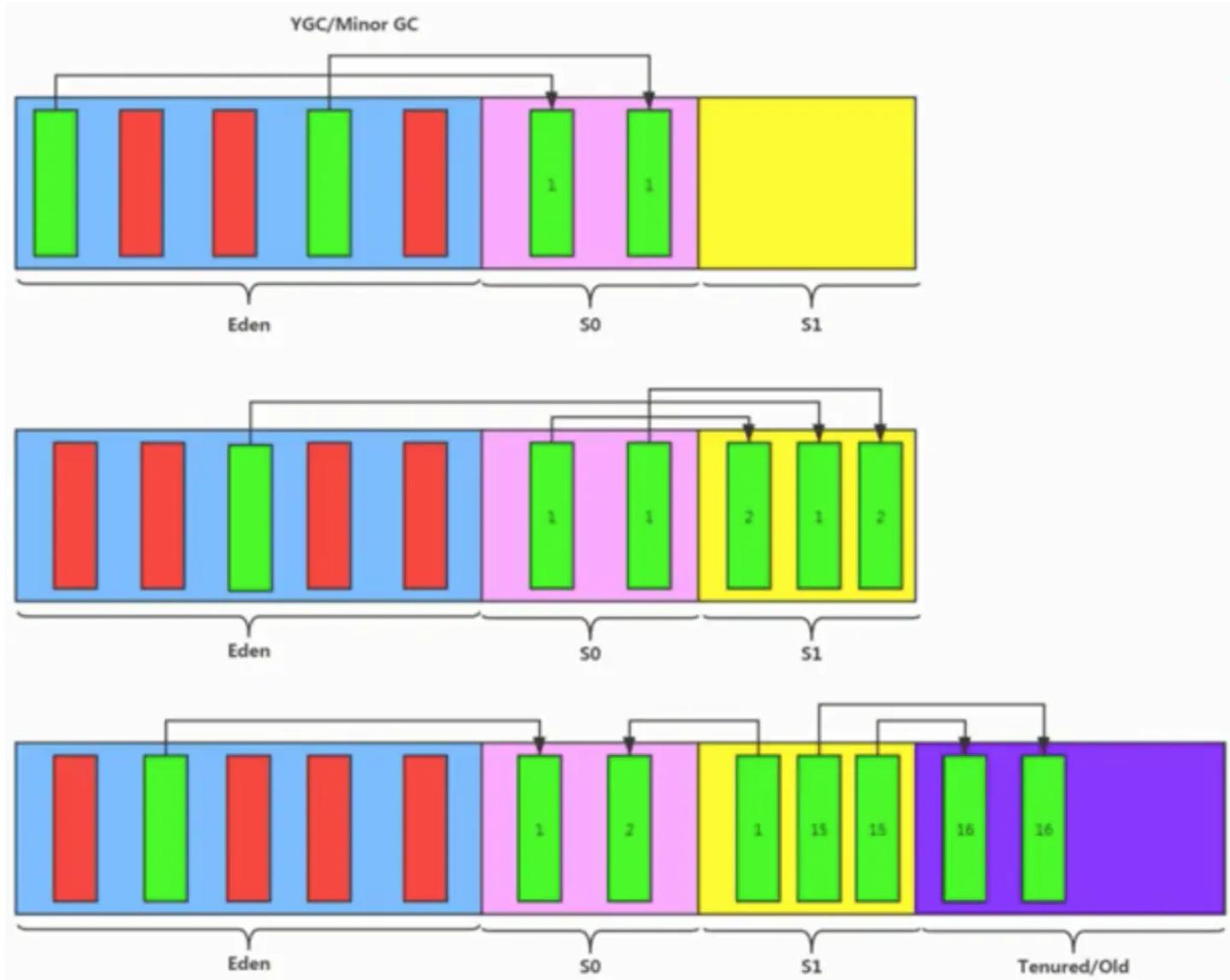
1. `new`的对象先放伊甸园区。此区有大小限制。
2. 当伊甸园的空间填满时，程序又需要创建对象，JVM的垃圾回收器将对伊甸园区进行垃圾回收 (Minor GC)，将伊甸园区中的不再被其他对象所引用的对象进行销毁。再加载新的对象放到伊甸园区
3. 然后将伊甸园中的剩余对象移动到幸存者0区。
4. 如果再次触发垃圾回收，此时上次幸存下来的放到幸存者0区的，如果没有回收，就会放到幸存者1区。
5. 如果再次经历垃圾回收，此时会重新放回幸存者0区，接着再去幸存者1区。

6. 啥时候能去养老区呢？可以设置次数。默认是15次。·可以设置参数： -XX:MaxTenuringThreshold 进行设置。
7. 在养老区，相对悠闲。当老年区内存不足时，再次触发GC： Major GC，进行养老区的内存清理。
8. 若养老区执行了Major GC之后发现依然无法进行对象的保存，就会产生OOM异常。

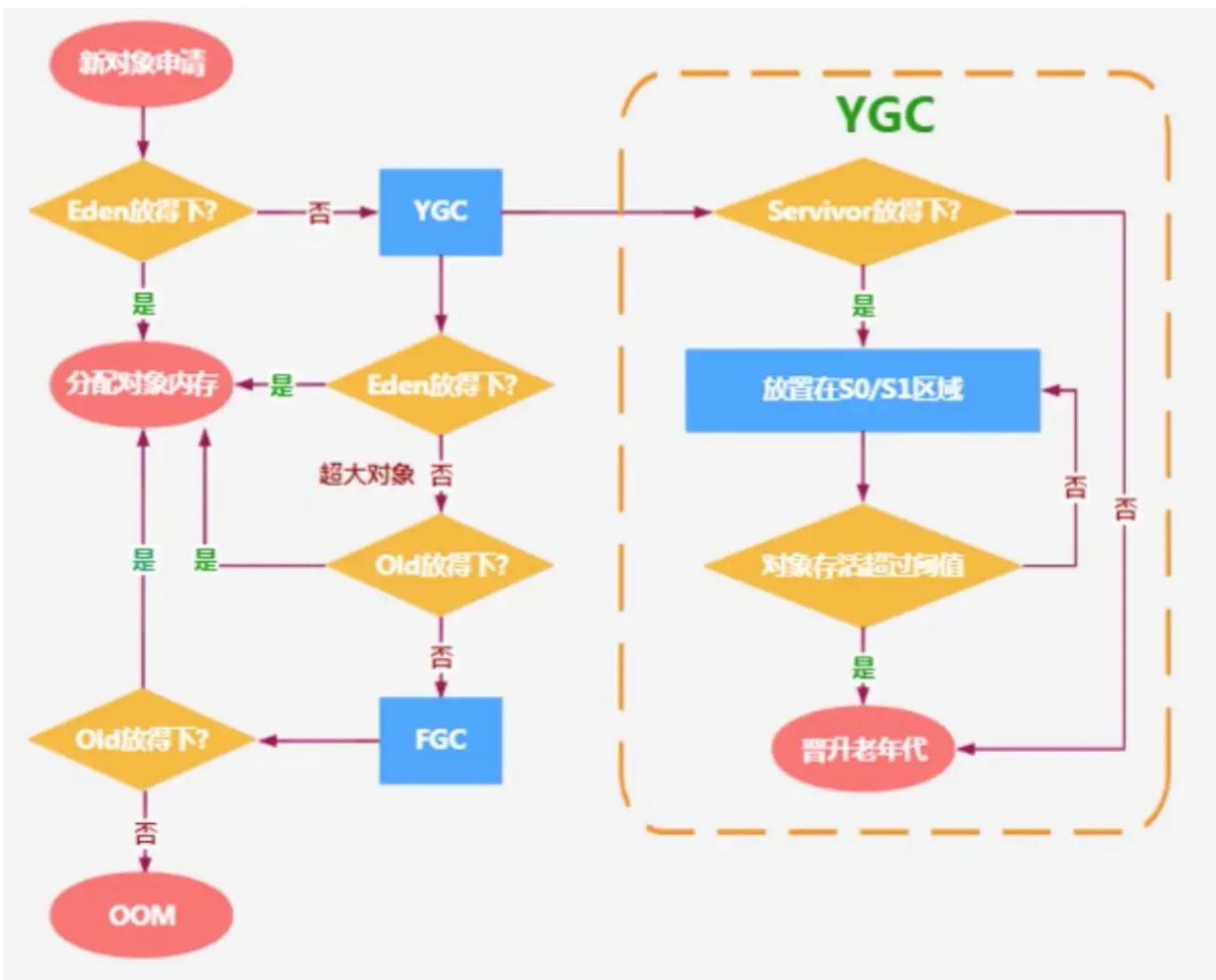
总结

针对幸存者s0,s1区：复制之后有交换，谁空谁是to

关于垃圾回收：频繁在新生区收集，很少在养老区收集，几乎不再永久区/元空间收集。



4.2 对象分配的特殊情况



5. Minor GC、Major GC、Full GC

JVM在进行GC时，并非每次都针对上面三个内存区域（新生代、老年代、方法区）一起回收的，大部分时候回收都是指新生代。

针对hotSpot VM的实现，它里面的GC按照回收区域又分为两大种类型：一种是部分收集（Partial GC），一种是整堆收集（Full GC）

- 部分收集：不是完整收集整个Java堆的垃圾收集。其中又分为：
 - 新生代收集（Minor GC / Young GC）：只是新生代的垃圾收集
 - 老年代收集（Major GC / Old GC）：只是老年代的垃圾收集
 - 目前，只有 CMS GC 会有单独收集老年代的行为
 - 注意，很多时候 Major GC 会和 Full GC 混淆使用，需要具体分辨是老年代回收还是整堆回收
 - 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
 - 目前，只有 G1 GC 会有这种行为

- 整堆收集（**Full GC**）：收集整个java堆和方法区的垃圾收集
- 年轻代GC（**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** 区满，**Survivor** 满不会引发GC。(每次 **Minor GC** 会清理年轻代的内存，**Survivor** 是被动GC，不会主动GC)

因为Java对象大多都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，这一定义既清晰又利于理解。

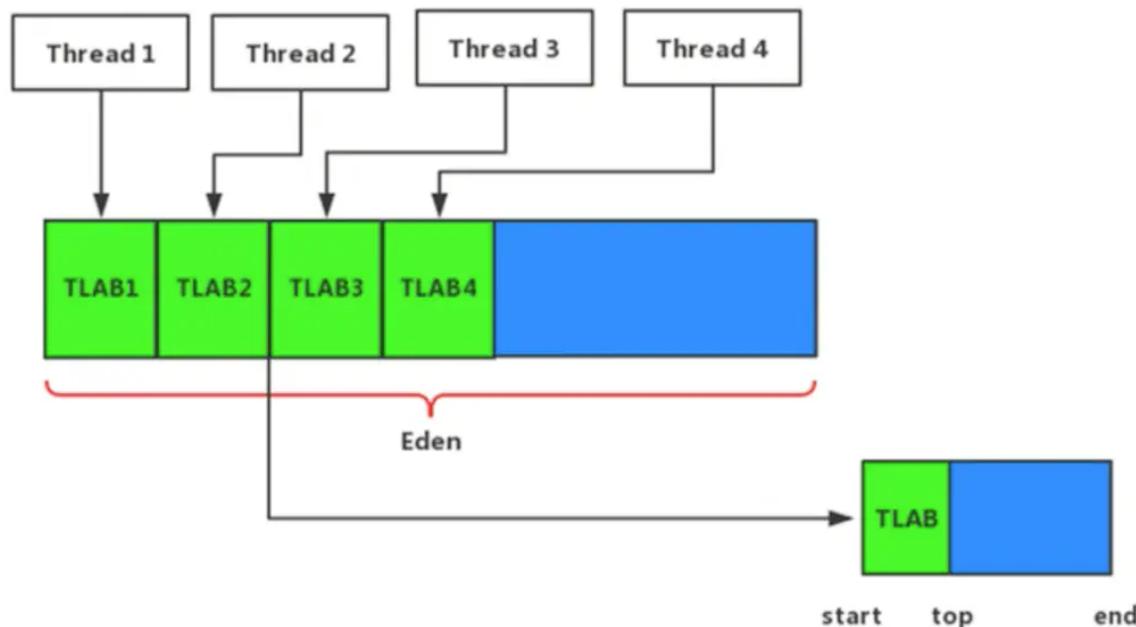
Minor GC 会引发 **STW**（**Stop the World**），暂停其他用户的线程，等垃圾回收结束，用户线程才恢复运行。
- 老年代GC(**Major GC / Full GC**)触发机制
 - 指发生在老年代的GC,对象从老年代消失时，就会触发 **Major GC** 或者 **Full GC**
 - 出现了 **Major GC**，经常会伴随至少一次的 **Minor GC**（不是绝对的，在 **Parallel Scavenge** 收集器的收集策略里就有直接进行**Major GC**的策略选择过程）
 - 也就是老年代空间不足时，会先尝试触发 **Minor GC**。如果之后空间还不足，则触发 **Major GC**
 - **Major GC** 速度一般会比 **Minor GC** 慢10倍以上，**STW** 时间更长
 - 如果**Major GC**后，内存还不足，就报OOM了
- **Full GC** 触发机制
 - 触发 **Full GC** 执行的情况有以下五种
 1. 调用 **System.gc()** 时，系统建议执行 **Full GC**，但是不必然执行
 2. 老年代空间不足
 3. 方法区空间不足
 4. 通过 **Minor GC** 后进入老年代的平均大小小于老年代的可用内存
 5. 由**Eden**区，**Survivor S0 (from)** 区向 **S1 (to)** 区复制时，对象大小由于**To Space**可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小
 - 说明：**Full GC** 是开发或调优中尽量要避免的，这样暂停时间会短一些

6. 内存分配策略

- 如果对象在**Eden**出生并经过第一次**Minor GC**后依然存活，并且能被**Survivor**容纳的话，将被移动到**Survivor**空间中，把那个对象年龄设为1.对象在**Survivor**区中每熬过一次**MinorGC**，年龄就增加一岁，当它的年龄增加到一定程度（默认15岁，其实每个JVM、每个GC都有所不同）时，就会被晋升到老年代中
 - 对象晋升老年代的年龄阈值，可以通过选项 **-XX: MaxTenuringThreshold**来设置
- 针对不同年龄段的对象分配原则如下：
 - 优先分配到**Eden**
 - 大对象直接分配到老年代
 - 尽量避免程序中出现过多的大对象
 - 长期存活的对象分配到老年代
 - 动态对象年龄判断

- 如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入到老年带。无需等到MaxTenuringThreshold中要求的年龄
 - 空间分配担保
 - -XX:HandlePromotionFailure

7. 为对象分配内存：TLAB（线程私有缓存区域）



为什么有TLAB (Thread Local Allocation Buffer)

- 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，进而影响分配速度

什么是TLAB

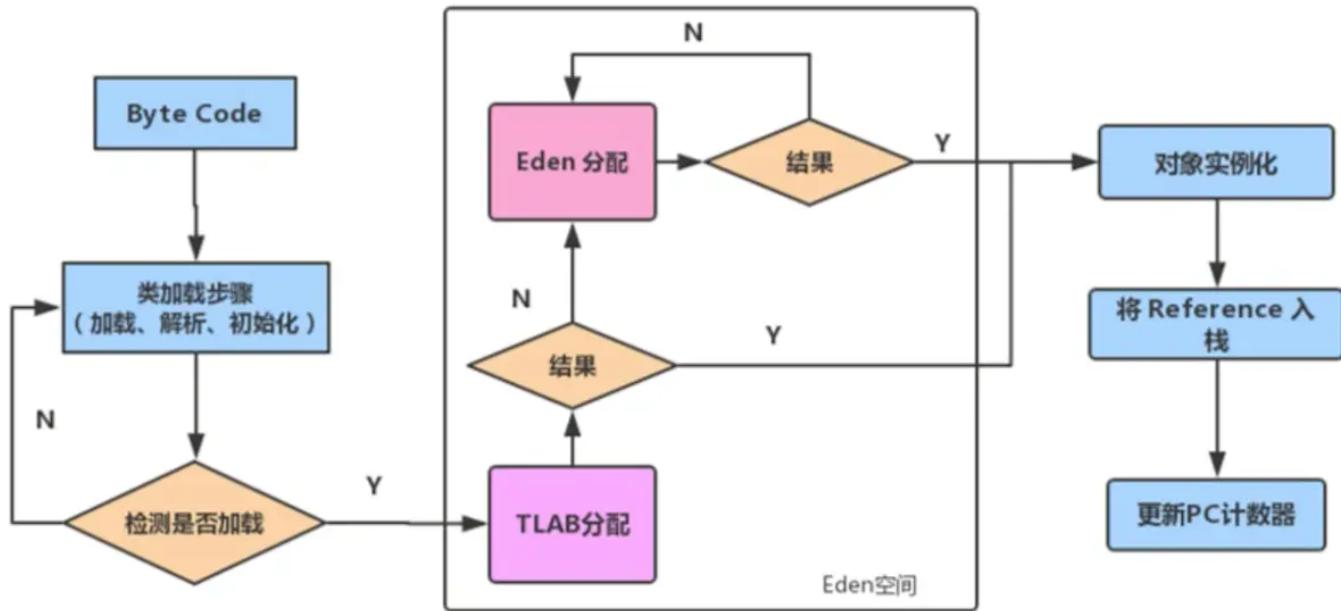
- 从内存模型而不是垃圾收集的角度，对Eden区域继续进行划分，JVM为每个线程分配了一个私有缓存区域，它包含在Eden空间内
- 多线程同时分配内存时，使用TLAB可以避免一系列的非线程安全问题，同时还能够提升内存分配的吞吐量，因此我们可以将这种内存分配方式称之为快速分配策略
- 所有OpenJDK衍生出来的JVM都提供了TLAB的设计

说明

- 尽管不是所有的对象实例都能够在TLAB中成功分配内存，但JVM明确是将TLAB作为内存分配的首选
- 在程序中，开发人员可以通过选项“-XX:UseTLAB”设置是否开启TLAB空间

- 默认情况下，TLAB空间的内存非常小，仅占有整个EDen空间的1%，当然我们可以通过选项"-XX:TLABWasteTargetPercent"设置TLAB空间所占用Eden空间的百分比大小
- 一旦对象在TLAB空间分配内存失败时，JVM就会尝试着通过使用加锁机制确保数据操作的原子性，从而直接在Eden空间中分配了内存

TLAB对象分配过程



8. 小结堆空间的参数设置

- 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（可能会存在修改，不再是初始值）
 - 具体查看某个参数的指令：
 - jps:** 查看当前运行中的进程
 - jinfo -flag SurvivorRatio 进程id:** 查看新生代中Eden和S0/S1空间的比例
- Xms:** 初始堆空间内存（默认为物理内存的1/64）
- Xmx:** 最大堆空间内存（默认为物理内存的1/4）
- Xmn:** 设置新生代大小（初始值及最大值）
- XX:NewRatio:** 配置新生代与老年代在堆结构的占比
- 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(默认15)
- XX:+PrintGCDetails:** 输出详细的GC处理日志
 - 打印gc简要信息：① **-XX:+PrintGC** ② **-verbose:gc**
- XX:HandlePromotionFailure:** 是否设置空间分配担保

说明

在发生Minor Gc之前，虚拟机会检查老年代最大可用的连续空间是否大于新生代所有对象的总空间。

- 如果大于，则此次Minor GC是安全的
- 如果小于，则虚拟机会查看-XX:HandlePromotionFailure设置值是否允许担保失败。（JDK 7以后的规则HandlePromotionFailure可以认为就是true）
 - 如果HandlePromotionFailure=true,那么会继续检查老年代最大可用连续空间是否大于历次晋升到老年代的对象的平均大小。
 - ✓如果大于，则尝试进行一次Minor GC,但这次Minor GC依然是有风险的;
 - ✓如果小于，则改为进行一次Full GC。
 - 如果HandlePromotionFailure=false,则改为进行一次Full GC。

在JDK6 Update24之后（JDK7），HandlePromotionFailure参数不会再影响到虚拟机的空间分配担保策略，观察openJDK中的源码变化，虽然源码中还定义了HandlePromotionFailure参数，但是在代码中已经不会再使用它。JDK6 Update24之后的规则变为只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小就会进行Minor GC,否则将进行Full GC。

9.堆是分配对象的唯一选择么（不是）

在《深入理解Java虚拟机》中关于Java堆内存有这样一段描述：随着JIT编译期的发展与逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。

在Java虚拟机中，对象是在Java堆中分配内存的，这是一个普遍的常识。但是，有一种特殊情况，那就是如果经过逃逸分析（Escape Analysis）后发现，一个对象并没有逃逸出方法的话，那么就可能被优化成栈上分配。这样就无需在堆上分配内存，也无须进行垃圾回收了。这也是最常见的堆外存储技术。

此外，前面提到的基于OpenJDK深度定制的TaoBaoVM,其中创新的GCIH(GCinvisble heap)技术实现off-heap,将生命周期较长的Java对象从heap中移至heap外，并且GC不能管理GCIH内部的Java对象，以此达到降低GC的回收频率和提升GC的回收效率的目的。

- 如何将堆上的对象分配到栈，需要使用逃逸分析手段。
- 这是一种可以有效减少Java程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法。
- 通过逃逸分析，Java Hotspot编译器能够分析出一个新的对象的引用的使用范围从而决定是否要将这个对象分配到堆上。
- 逃逸分析的基本行为就是分析对象动态作用域：
 - 当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有发生逃逸。
 - 当一个对象在方法中被定义后，它被外部方法所引用，则认为发生逃逸。例如作为调用参数传递到其他地方中。
- 如何快速的判断是否发生了逃逸分析，就看new的对象实体是否有可能在方法外被调用

逃逸分析

```

/**
 * 逃逸分析
 *
 * 如何快速的判断是否发生了逃逸分析，就看new的对象实体是否有可能在方法外被调用。
 */
public class EscapeAnalysis {

    public EscapeAnalysis obj;

    /*
    方法返回EscapeAnalysis对象，发生逃逸
    */
    public EscapeAnalysis getInstance(){
        return obj == null? new EscapeAnalysis() : obj;
    }

    /*
    为成员属性赋值，发生逃逸
    */
    public void setObj(){
        this.obj = new EscapeAnalysis();
    }

    //思考：如果当前的obj引用声明为static的？仍然会发生逃逸。

    /*
    对象的作用域仅在当前方法中有效，没有发生逃逸
    */
    public void useEscapeAnalysis(){
        EscapeAnalysis e = new EscapeAnalysis();
    }

    /*
    引用成员变量的值，发生逃逸
    */
    //比如此时去修改e的值可能会影响成员变量的值
    public void useEscapeAnalysis1(){
        EscapeAnalysis e = getInstance();
        //getInstance().xxx()同样会发生逃逸
    }
}

```

代码优化

使用逃逸分析，编译器可以对代码做如下优化：

1. 栈上分配：将堆分配转化为栈分配。如果一个对象在子线程中被分配，要使指向该对象的指针永远不会逃逸，对象可能是栈分配的候选，而不是堆分配
2. 同步省略：如果一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步
3. 分离对象或标量替换：有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

栈上分配

- JIT编译器在编译期间根据逃逸分析的结果，发现如果一个对象并没有逃逸出方法的话，就可能被优化成栈上分配。分配完成之后，继续在调用栈内执行，最后线程结束，栈空间被回收，局部变量对象也被回收。这样就无须机型垃圾回收了
- 常见的栈上分配场景：给成员变量赋值、方法返回值、实例引用传递

代码分析

以下代码，关闭逃逸分析（`-XX:-DoEscapeAnalysis`），维护10000000个对象，如果开启逃逸分析，只维护少量对象（JDK7 逃逸分析默认开启）

```
/**  
 * 栈上分配测试  
 * -Xmx1G -Xms1G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails  
 */  
  
public class StackAllocation {  
    public static void main(String[] args) {  
        long start = System.currentTimeMillis();  
  
        for (int i = 0; i < 10000000; i++) {  
            alloc();  
        }  
        // 查看执行时间  
        long end = System.currentTimeMillis();  
        System.out.println("花费的时间为: " + (end - start) + " ms");  
        // 为了方便查看堆内存中对象个数，线程sleep  
        try {  
            Thread.sleep(1000000);  
        } catch (InterruptedException e1) {  
            e1.printStackTrace();  
        }  
    }  
  
    private static void alloc() {  
        User user = new User(); //未发生逃逸  
    }  
  
    static class User {  
    }  
}
```

同步省略

- 线程同步的代价是相当高的，同步的后果是降低并发性和性能
- 在动态编译同步块的时候，JIT编译器可以借助逃逸分析来判断同步块所使用的锁对象是否只能被一个线程访问而没有被发布到其他线程。如果没有，那么JIT编译器在编译这个同步块的时候就会取

消对这部分代码的同步。这样就能大大提高并发性和性能。这个取消同步的过程就叫同步省略，也叫锁消除

```
/**  
 * 同步省略说明  
 */  
public class SynchronizedTest {  
    public void f() {  
        Object hollis = new Object();  
        synchronized(hollis) {  
            System.out.println(hollis);  
        }  
    }  
    //代码中对hollis这个对象进行加锁，但是hollis对象的生命周期只在f()方法中  
    //并不会被其他线程所访问控制，所以在JIT编译阶段就会被优化掉。  
    //优化为 ↓  
    public void f2() {  
        Object hollis = new Object();  
        System.out.println(hollis);  
    }  
}
```

分离对象或标量替换

- 标量**Scalar**是指一个无法在分解成更小的数据的数据。Java中的原始数据类型就是标量
- 相对的，那些还可以分解的数据叫做聚合量(**Aggregate**)，Java中对象就是聚合量，因为它可以分解成其他聚合量和标量
- 在JIT阶段，如果经过逃逸分析，发现一个对象不会被外界访问的话，那么经过JIT优化，就会把这个对象拆解成若干个其中包含的若干个成员变量来替代。这个过程就是标量替换

```
public class ScalarTest {  
    public static void main(String[] args) {  
        alloc();  
    }  
    public static void alloc(){  
        Point point = new Point(1,2);  
    }  
}  
class Point{  
    private int x;  
    private int y;  
    public Point(int x,int y){  
        this.x = x;  
        this.y = y;  
    }  
}
```

以上代码，经过标量替换后，就会变成

```
public static void alloc(){
    int x = 1;
    int y = 2;
}
```

可以看到，**Point**这个聚含量经过逃逸分析后，发现他并没有逃逸，就被替换成两个标量了。那么标量替换有什么好处呢？就是可以大大减少堆内存的占用。因为一旦不需要创建对象了，那么就不再需要分配堆内存了。

标量替换为栈上分配提供了很好的基础。

逃逸分析小结

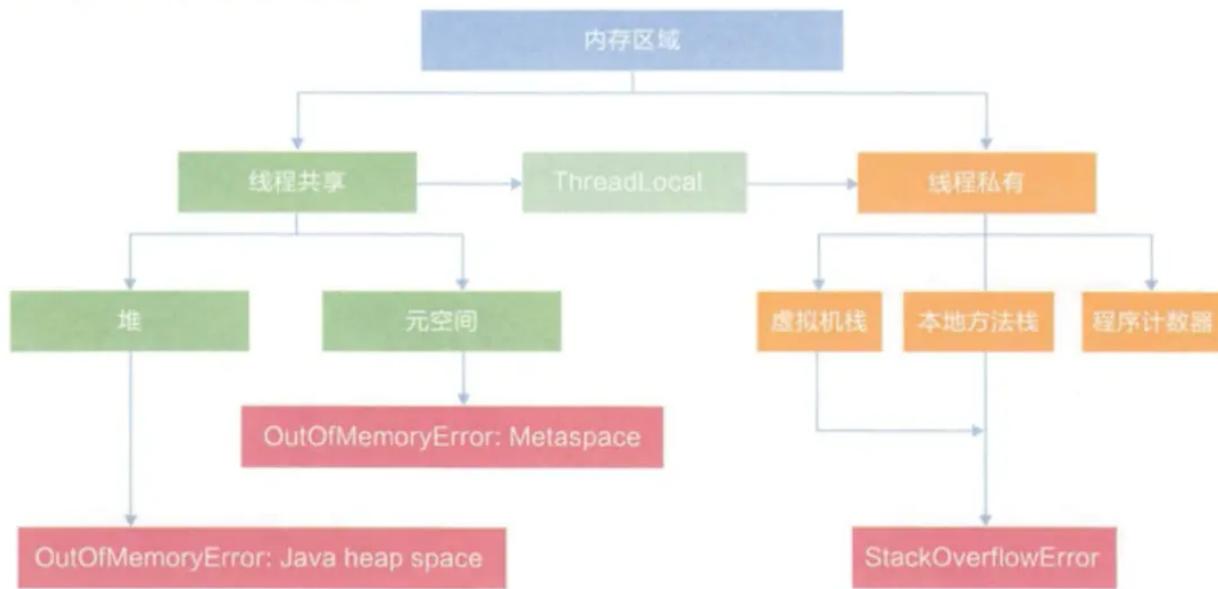
- 关于逃逸分析的论文在**1999**年就已经发表了，但直到**JDK1.6**才有实现，而且这项技术到如今也并不是十分成熟的。
- 其根本原因就是无法保证逃逸分析的性能消耗一定能高于他的消耗。虽然经过逃逸分析可以做标量替换、栈上分配、和锁消除。但是逃逸分析自身也是需要进行一系列复杂的分析的，这其实也是一个相对耗时的过程。
- 一个极端的例子，就是经过逃逸分析之后，发现没有一个对象是不逃逸的。那这个逃逸分析的过程就白白浪费掉了。
- 虽然这项技术并不十分成熟，但是它也是即时编译器优化技术中一个十分重要的手段。
- 注意到有一些观点，认为通过逃逸分析，**JVM**会在栈上分配那些不会逃逸的对象，这在理论上是可行的，但是取决于**JVM**设计者的选择。据我所知，**Oracle HotspotJVM**中并未这么做，这一点在逃逸分析相关的文档里已经说明，所以可以明确所有的对象实例都是创建在堆上。
- 目前很多书籍还是基于**JDK7**以前的版本，**JDK**已经发生了很大变化，**intern**字符串的缓存和静态变量曾经都被分配在永久代上，而永久代已经被元数据区取代。但是，**intern**字符串缓存和静态变量并不是被转移到元数据区，而是直接在堆上分配，所以这一点同样符合前面一点的结论：对象实例都是分配在堆上。
- 年轻代是对象的诞生、生长、消亡的区域，一个对象在这里产生、应用、最后被垃圾回收器收集、结束生命
- 老年代防止长生命周期对象，通常都是从**Survivor**区域筛选拷贝过来的**Java**对象。当然，也有特殊情况，我们知道普通的对象会被分配在**TLAB**上，如果对象较大，**JVM**会试图直接分配在**Eden**其他位置上；如果对象被打，完全无法在新生代找到足够长的连续空闲空间，**JVM**就会直接分配到老年代
- 当GC只发生在年轻代中，回收年轻对象的行为被称为**MinorGC**。当GC发生在老年代时则被称为**MajorGC**或者**FullGC**。一般的，**MinorGC**的发生频率要比**MajorGC**高很多，即老年代中垃圾回收发生的频率大大低于年轻代

运行时数据区**3**-方法区

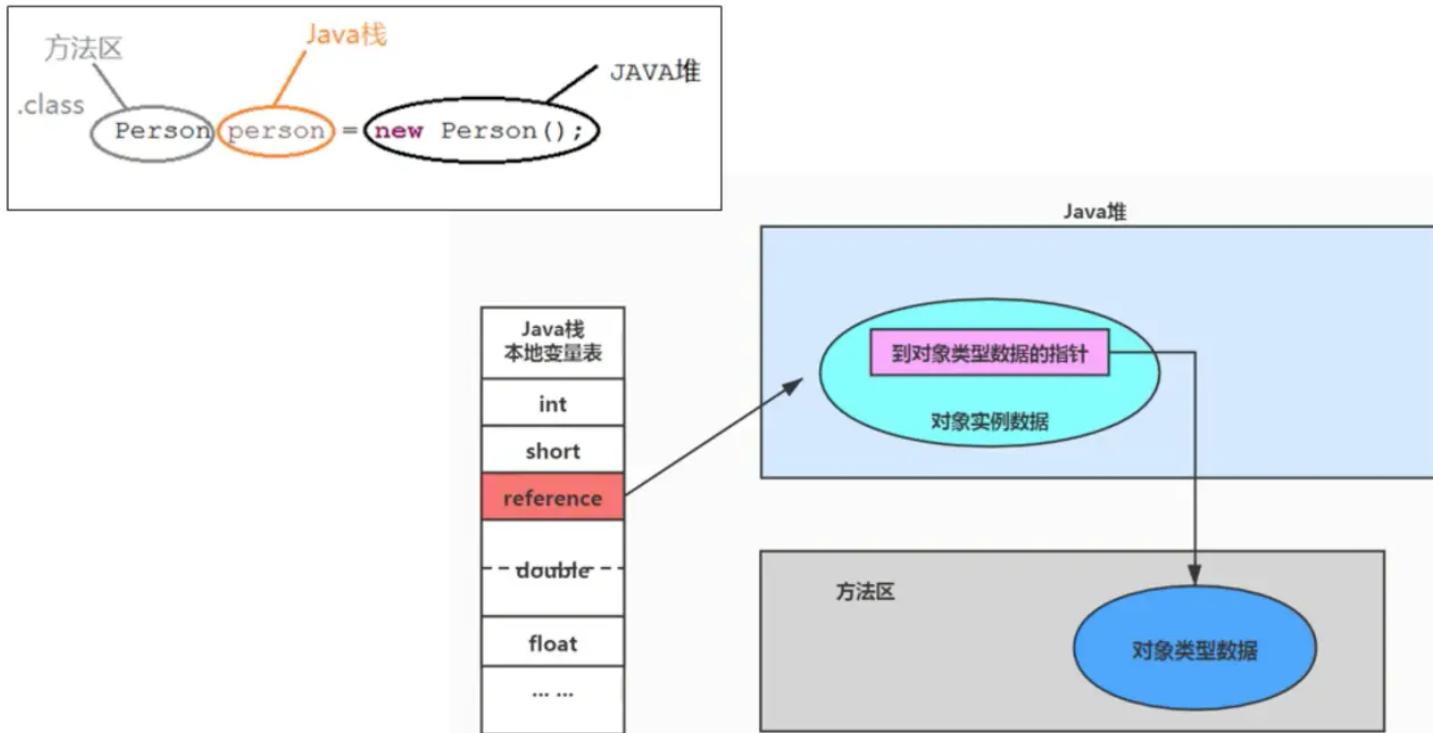
1. 堆、栈、方法区的交互关系

运行时数据区结构图

从线程共享与否的角度来看



堆、栈、方法区的交互关系



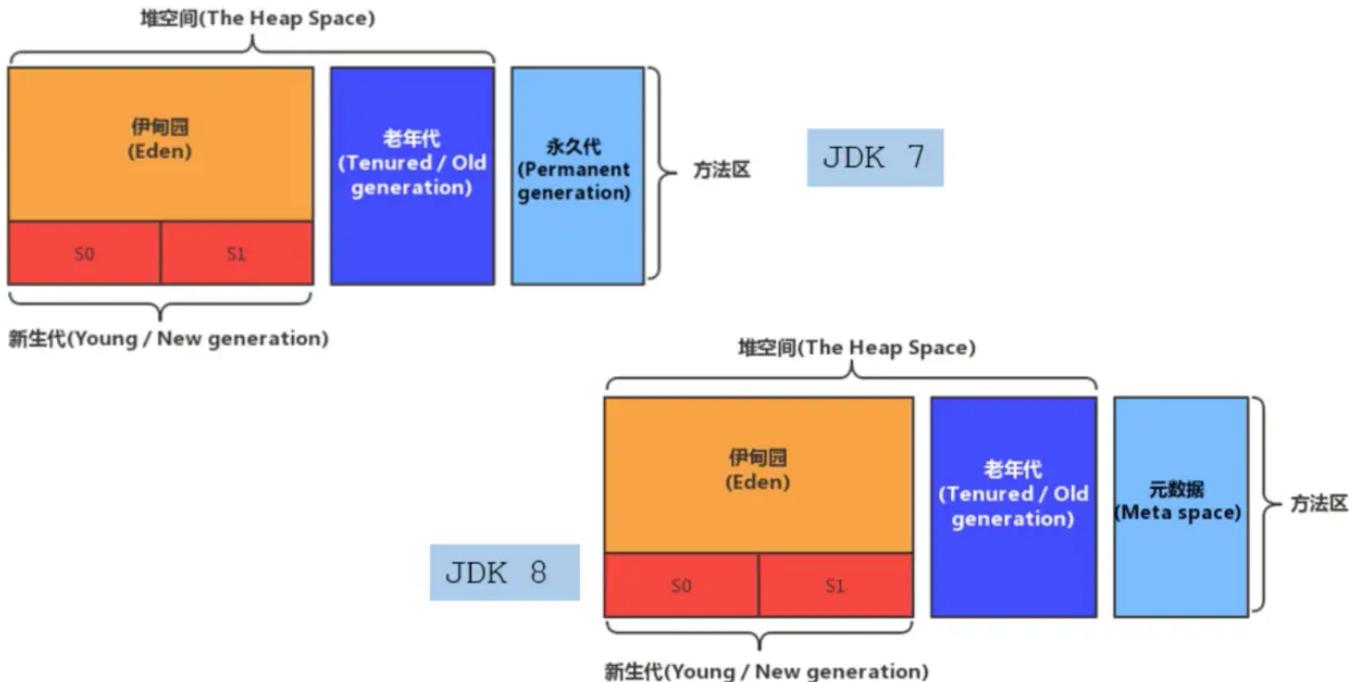
2. 方法区的理解

《Java虚拟机规范》中明确说明：‘尽管所有的方法区在逻辑上属于堆的一部分，但一些简单的实现可能不会选择去进行垃圾收集或者进行压缩。’但对于HotSpotJVM而言，方法区还有一个别名叫做Non-heap（非堆），目的就是要和堆分开。

所以，方法区可以看作是一块独立于Java堆的内存空间。

- 方法区（Method Area）与Java堆一样，是各个线程共享的内存区域
 - 方法区在JVM启动时就会被创建，并且它的实际的物理内存空间中和Java堆区一样都可以是不连续的
 - 方法区的大小，跟堆空间一样，可以选择固定大小或者可拓展
 - 方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出错误：java.lang.OutOfMemoryError:PermGen space 或者 java.lang.OutOfMemoryError:Metaspace，比如：
 - 加载大量的第三方jar包；
 - Tomcat部署的工程过多；
 - 大量动态生成反射类；
 - 关闭JVM就会释放这个区域的内存
- 例，使用jvisualvm查看加载类的个数
- 在jdk7及以前，习惯上把方法区称为永久代。jdk8开始，使用元空间取代了永久代
 - 本质上，方法区和永久代并不等价。仅是对hotSpot而言的。《java虚拟机规范》对如何实现方法区，不做统一要求。例如：BEA JRockit/IBM J9中不存在永久代的概念
 - 现在看来，当年使用永久代，不是好的idea。导致Java程序更容易OOM(超过-XX:MaxPermSize上限)

方法区在jdk7及jdk8的落地实现



- 在jdk8中，终于完全废弃了永久代的概念，改用与JRockit、J9一样在本地内存中实现的元空间（Metaspace）来代替
- 元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代最大的区别在于：元空间不再虚拟机设置的内存中，而是使用本地内存
- 永久代、元空间不只是名字变了。内部结构也调整了
- 根据《Java虚拟机规范》得规定，如果方法区无法满足新的内存分配需求时，将抛出OOM异常。

3. 设置方法区大小与OOM

方法区的大小不必是固定的，jvm可以根据应用的需要动态调整。

jdk7及以前：

- 通过-XX: PermSize来设置永久代初始分配空间。默认值是20.75M
- -XX : MaxPermSize来设定永久代最大可分配空间。32位机器默认是64M, 64位机器模式是82M
- 当JVM加载的类信息容量超过了这个值，会报异常OutOfMemoryError : PermGen space

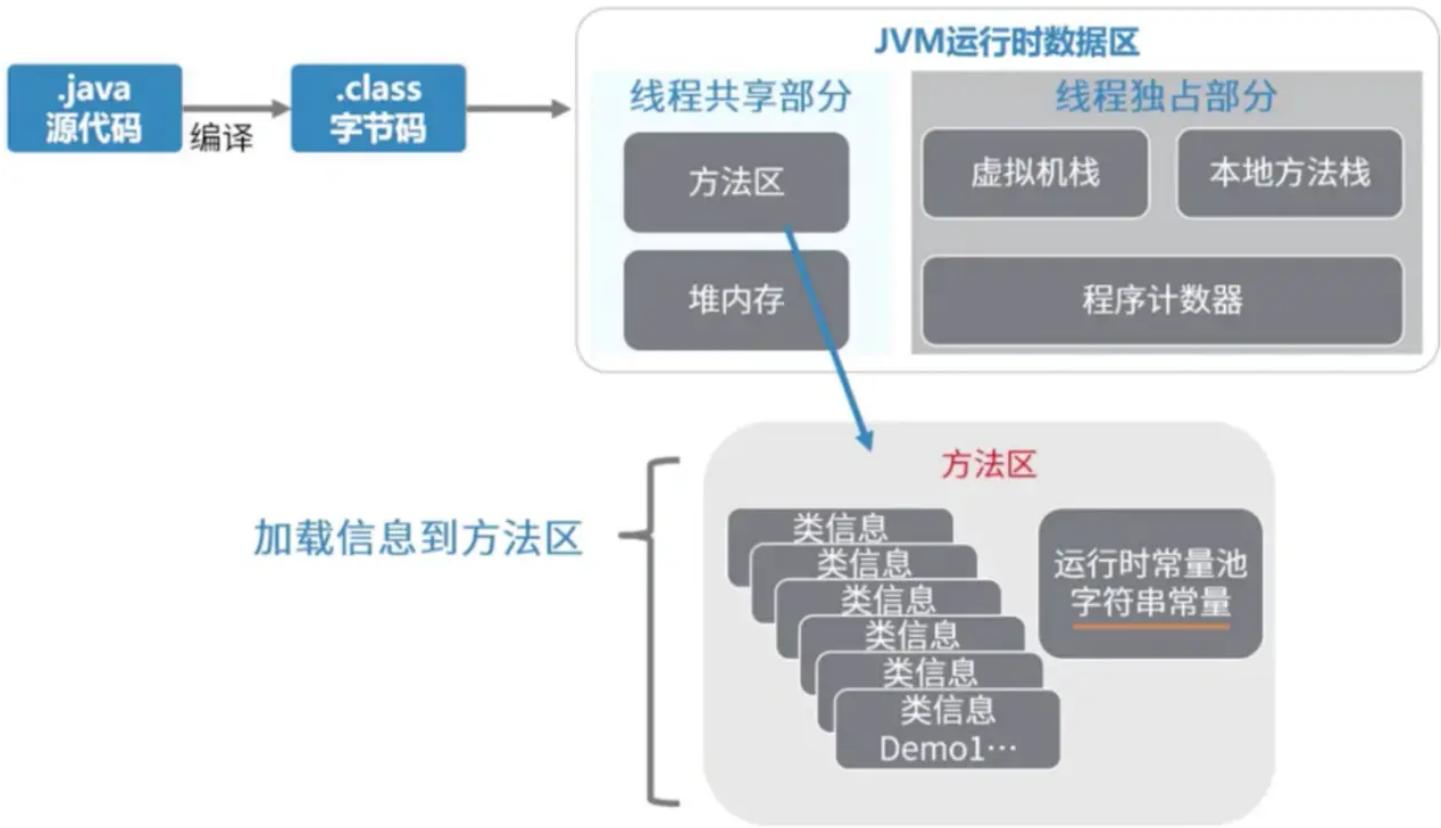
jdk8及以后：

- 元数据区大小可以使用参数-XX: MetaspaceSize和-XX : MaxMetaspaceSize指定，替代上述原有的两个参数。
- 默认值依赖于平台。windows下，-XX: MetaspaceSize是21M，-XX: MaxMetaspaceSize的值是-1，即没有限制。||
- 与永久代不同，如果不指定大小，默认情况下，虚拟机会耗尽所有的可用系统内存。如果元数据区发生溢出，虚拟机一样会抛出异常OutOfMemoryError: Metaspace
- -XX: MetaspaceSize: 设置初始的元空间大小。对于一个64位的服务器端JVM来说，其默认的-XX: MetaspaceSize值为21MB.这就是初始的高水位线，一旦触及这个水位线，Full GC将会被触发并卸载没用的类（即这些类对应的类加载器不再存活），然后这个高水位线将会重置。新的高水位线的值取决于GC后释放了多少元空间。如果释放的空间不足，那么在不超过MaxMetaspaceSize时，适当提高该值。如果释放空间过多，则适当降低该值。
- 如果初始化的高水位线设置过低，上述高水位线调整情况会发生很多次。通过垃圾回收器的日志可以观察到Full GC多次调用。为了避免频繁地GC，建议将-XX: MetaspaceSize设置为一个相对较高的值。

方法区OOM

1. 要解决OOM异常或heap space的异常，一般的手段是首先通过内存映像分析工具（如Eclipse Memory Analyzer）对dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏（Memory Leak）还是内存溢出（Memory Overflow）。
2. 如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots 的引用链。于是就能找到泄漏对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息，以及GC Roots引用链的信息，就可以比较准确地定位出泄漏代码的位置。
3. 如果不存在内存泄漏，换句话说就是内存中的对象确实都还必须存活，那就应当检查虚拟机的堆参数（-Xmx与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

4. 方法区的内部结构



《深入理解Java虚拟机》书中对方法区存储内容描述如下：它用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等。



类型信息

对每个加载的类型（类class、接口interface、枚举enum、注解annotation），JVM必须在方法区中存储以下类型信息：

1. 这个类型的完整有效名称（全名=包名.类名）
2. 这个类型直接父类的完整有效名（对于interface或是java.lang.Object，都没有父类）
3. 这个类型的修饰符（public, abstract, final的某个子集）
4. 这个类型直接接口的一个有序列表

域信息（成员变量）

- JVM必须在方法区中保存类型的所有域的相关信息以及域的声明顺序。
- 域的相关信息包括：域名称、域类型、域修饰符（public, private, protected, static, final, volatile, transient的某个子集）

方法信息

JVM必须保存所有方法的以下信息，同域信息一样包括声明顺序：

- 方法名称
- 方法的返回类型（或void）
- 方法参数的数量和类型（按顺序）
- 方法的修饰符（public, private, protected, static, final, synchronized, native, abstract的一个子集）
- 方法的字节码（bytecodes）、操作数栈、局部变量表及大小（abstract和native方法除外）
- 异常表（abstract和native方法除外）
 - 每个异常处理的开始位置、结束位置、代码处理在程序计数器中的偏移地址、被捕获的异常类的常量池索引

non-final的类变量

- 静态变量和类关联在一起，随着类的加载而加载，他们成为类数据在逻辑上的一部分
- 类变量被类的所有实例所共享，即使没有类实例你也可以访问它。

以下代码不会报空指针异常

```
public class MethodAreaTest {  
    public static void main(String[] args) {  
        Order order = null;  
        order.hello();  
        System.out.println(order.count);  
    }  
}  
  
class Order {  
    public static int count = 1;  
    public static final int number = 2;  
  
    public static void hello() {  
        System.out.println("hello!");  
    }  
}
```

全局常量 static final

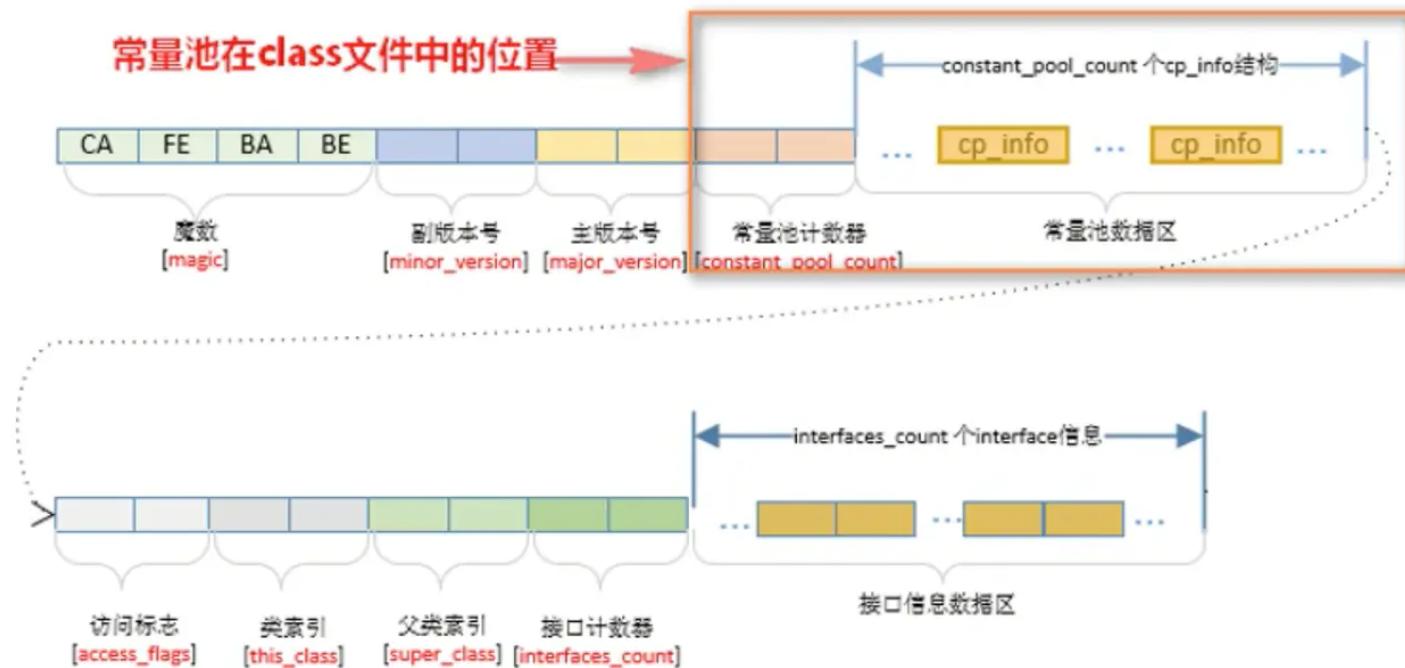
被声明为final的类变量的处理方法则不同，每个全局常量在编译的时候就被分配了。

代码解析

Order.class字节码文件，右键Open in Terminal打开控制台，使用javap -v -p Order.class > tst.txt 将字节码文件反编译并输出为txt文件，可以看到被声明为static final的常量number在编译的时候就被赋值了，这不同于没有被final修饰的static变量count是在类加载的准备阶段被赋值

运行时常量池

常量池



- 一个有效的字节码文件中除了包含类的版本信息、字段、方法以及接口等描述信息外，还包含一项信息那就是常量池表（Constant Pool Table），包括各种字面量和对类型域和方法的符号引用。
- 一个 java 源文件中的类、接口，编译后产生一个字节码文件。而 Java 中的字节码需要数据支持，通常这种数据会很大以至于不能直接存到字节码里，换另一种方式，可以存到常量池这个字节码包含了指向常量池的引用。在动态链接的时候会用到运行时常量池。
- 比如如下代码，虽然只有 194 字节，但是里面却使用了 string、System、Printstream 及 Object 等结构。这里代码量其实已经很小了。如果代码多，引用到的结构会更多！

几种在常量池内存储的数据类型包括：

- 数量值
- 字符串值
- 类引用
- 字段引用
- 方法引用

小结

常量池，可以看做是一张表，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。

运行时常量池

- 运行时常量池（Runtime Constant Pool）是方法区的一部分。

- 常量池表（Constant Pool Table）是Class文件的一部分，用于存放编译期生成的各种字面量与符号引用，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。
- 运行时常量池，在加载类和接口到虚拟机后，就会创建对应的运行时常量池。
- JVM为每个已加载的类型（类或接口）都维护一个常量池。池中的数据项像数组项一样，是通过索引访问的。
- 运行时常量池中包含多种不同的常量，包括编译期就已经明确的数值字面量，也包括到运行期解析后才能够获得的方法或者字段引用。此时不再是常量池中的符号地址了，这里换为真实地址。
 - 运行时常量池，相对于Class文件常量池的另一重要特征是：具备动态性。

String.intern()

- 运行时常量池类似于传统编程语言中的符号表（symbol table），但是它所包含的数据却比符号表要更加丰富一些。
- 当创建类或接口的运行时常量池时，如果构造运行时常量池所需的内存空间超过了方法区所能提供的最大值，则JVM会抛OutOfMemoryError异常。

5.方法区的使用举例

```
public class MethodAreaDemo {
    public static void main(String[] args) {
        int x = 500;
        int y = 100;
        int a = x / y;
        int b = 50;
        System.out.println(a + b);
    }
}
```

main方法的字节码指令

```
0 sipush 500
3 istore_1
4 bipush 100
6 istore_2
7 iload_1
8 iload_2
9 idiv
10 istore_3
11 bipush 50
13 istore_4
15 getstatic #2 <java/lang/System.out>
18 iload_3
19 iload_4
21 iadd
22 invokevirtual #3 <java/io/PrintStream.println>
25 return
```

方法区 程序入口：main方法

| 序号 | 字节码指令 |
|----|------------------|
| 0 | sipush 500 |
| 3 | istore_1 |
| 4 | bipush 100 |
| 6 | istore_2 |
| 7 | iload_1 |
| 8 | iload_2 |
| 9 | idiv |
| 10 | istore_3 |
| 11 | bipush 50 |
| 13 | istore 4 |
| 15 | getstatic #2 |
| 18 | iload_3 |
| 19 | iload 4 |
| 21 | iadd |
| 22 | invokevirtual #3 |
| 25 | return |

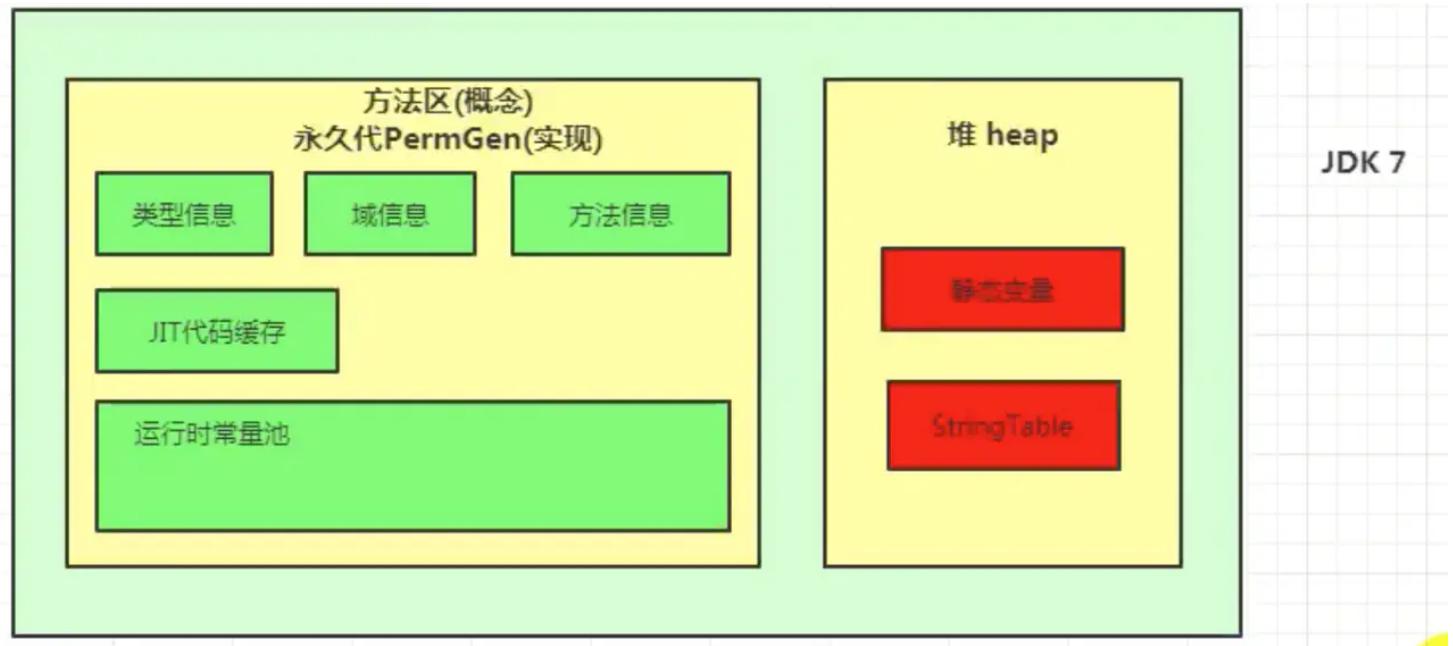
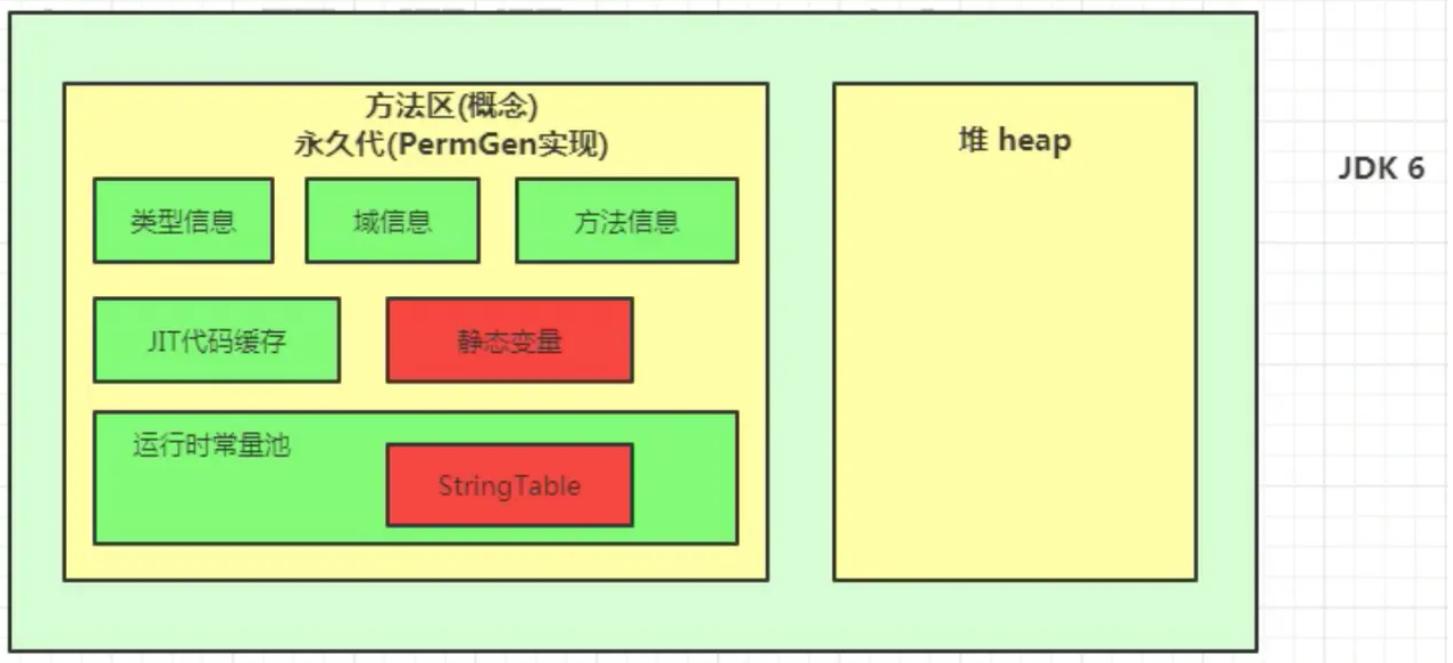
将栈顶两int类型数相加，结果入栈

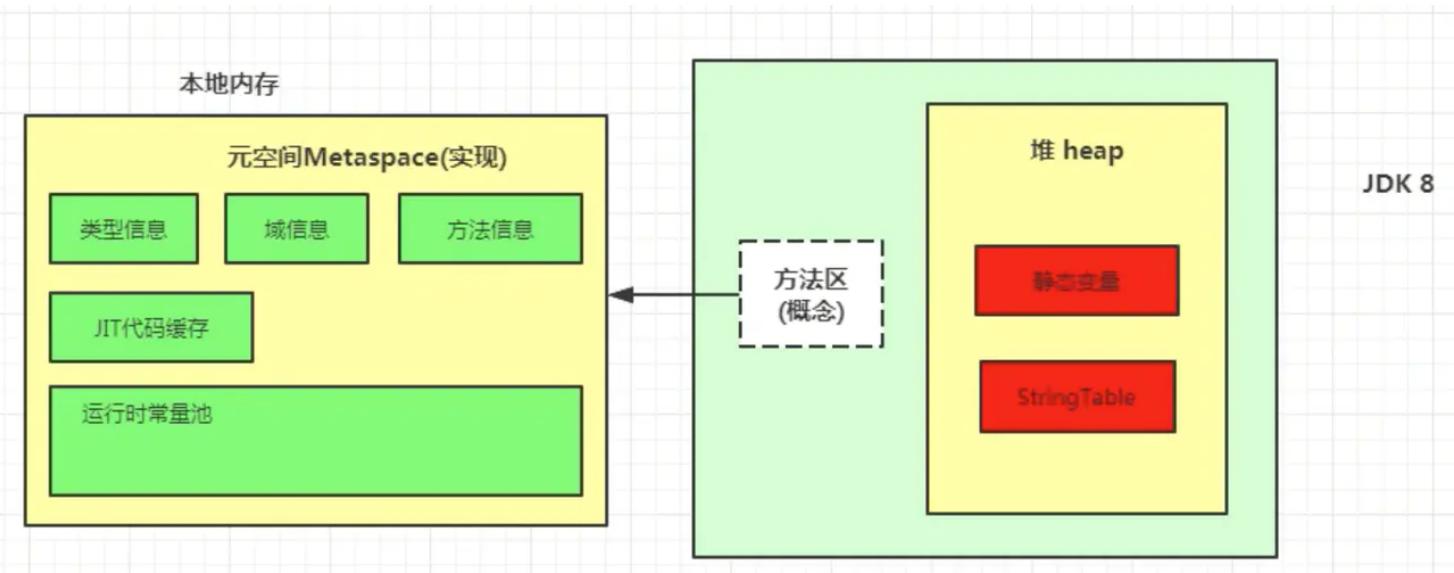


6.方法区的演进细节

- 首先明确：只有HotSpot才有永久代。BEA JRockit、IBM J9等来说，是不存在永久代的概念的。原则上如何实现方法区属于虚拟机实现细节，不受《Java虚拟机规范》管束，并不要求统一。
- Hotspot中方法区的变化：

- jdk1.6及之前：有永久代（permanent generation），静态变量存放在永久代上
- jdk1.7：有永久代，但已经逐步“去永久代”，字符串常量池、静态变量移除，保存在堆中
- jdk1.8及之后：无永久代，类型信息、字段、方法、常量保存在本地内存的元空间，但字符串常量池、静态变量仍在堆





永久代为什么要被元空间替换

- 随着Java8的到来，HotSpot VM中再也见不到永久代了。但是这并不意味着类的元数据信息也消失了。这些数据被移到了一个与堆不相连的本地内存区域，这个区域叫做元空间（Metaspace）。
- 由于类的元数据分配在本地内存中，元空间的最大可分配空间就是系统可用内存空间。
- 这项改动是很有必要的，原因有：
 - 为永久代设置空间大小是很难确定的。在某些场景下，如果动态加载类过多，容易产生Perm区的OOM。比如某个实际Web工程中，因为功能点比较多，在运行过程中，要不断动态加载很多类，经常出现致命错误。

```
"Exception in thread 'dubbo client x.x connector' java.lang.OutOfMemoryError: PermGen space"
"而元空间和永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。
```
 - 对永久代进行调优是很困难的。

StringTable 为什么要调整

jdk7中将StringTable放到了堆空间中。因为永久代的回收效率很低，在full gc的时候才会触发。而full GC是老年代的空间不足、永久代不足时才会触发。这就导致了StringTable回收效率不高。而我们开发中会有大量的字符串被创建，回收效率低，导致永久代内存不足。放到堆里，能及时回收内存.

如何证明静态变量存在哪

```

/**
 * 《深入理解Java虚拟机》中的案例：
 * staticObj、instanceObj、localObj存放在哪里？
 */
public class StaticObjTest {
    static class Test {
        static ObjectHolder staticObj = new ObjectHolder();
        ObjectHolder instanceObj = new ObjectHolder();

        void foo() {
            ObjectHolder localObj = new ObjectHolder();
            System.out.println("done");
        }
    }
}

private static class ObjectHolder {
}

public static void main(String[] args) {
    Test test = new StaticObjTest.Test();
    test.foo();
}
}

```

- 测试发现：三个对象的数据在内存中的地址都落在Eden区范围内，所以结论：只要是对象实例必然会在Java堆中分配。
- 接着，找到了一个引用该staticObj对象的地方，是在一个java.lang.Class的实例里，并且给出了这个实例的地址，通过Inspector查看该对象实例，可以清楚看到这确实是一个java.lang.Class类型的对象实例，里面有一个名为staticObj的实例字段：

7.方法区的垃圾回收

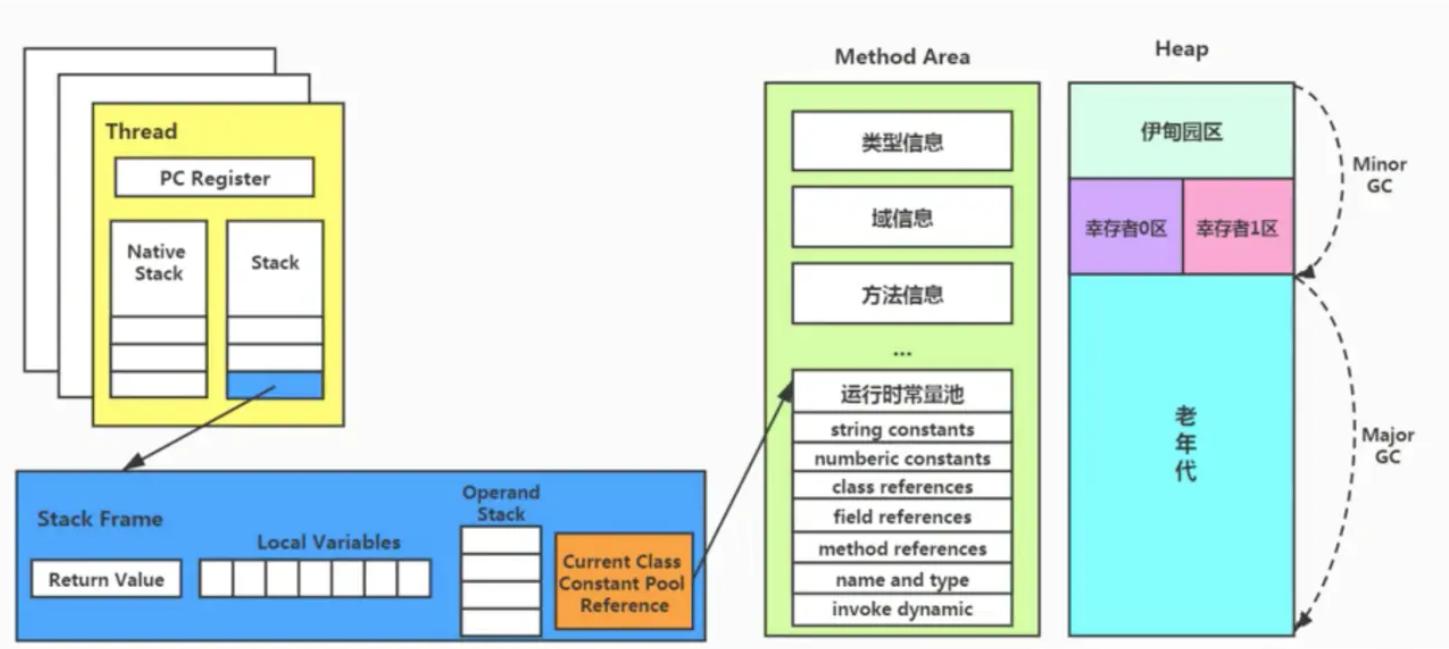
一般来说这个区域的回收效果比较难令人满意，尤其是类型的卸载，条件相当苛刻。但是这部分区域的回收有时又确实是必要的。以前 Sun 公司的 Bug 列表中，曾出现过的若干个严重的 Bug 就是由于低版本的 Hotspot 虚拟机对此区域未完全回收而导致内存泄漏。

方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不再使用的类型

- 先来说说方法区内常量池之中主要存放的两大类常量：字面量和符号引用。字面量比较接近Java语言层次的常量概念，如文本字符串、被声明为final的常量值等。而符号引用则属于编译原理方面的概念，包括下面三类常量：
 - 类和接口的全限定名
 - 字段的名称和描述符
 - 方法的名称和描述符
- HotSpot虚拟机对常量池的回收策略是很明确的，只要常量池中的常量没有被任何地方引用，就可以被回收。

- 回收废弃常量与回收Java堆中的对象非常类似。
- 判定一个常量是否“废弃”还是相对简单，而要判定一个类型是否属于“不再被使用的类”的条件就比较苛刻了。需要同时满足下面三个条件：
 1. 该类所有的实例都已经被回收，也就是Java堆中不存在该类及其任何派生子类的实例。
 2. 加载该类的类加载器已经被回收，这个条件除非是经过精心设计的可替换类加载器的场景，如OSGi、JSP的重加载等，否则通常是很难达成的。|】
 3. 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。
- Java虚拟机被允许对满足上述三个条件的无用类进行回收，这里说的仅仅是“被允许”，而并不是和对象一样，没有引用了就必然会回收。关于是否要对类型进行回收，HotSpot虚拟机提供了一Xnoclassgc参数进行控制，还可以使用 verbose: class以及XX: +TraceClassLoading、XX: +TraceClassUnLoading查看类加载和卸载信息
- 在大量使用反射、动态代理、CGLib等字节码框架，动态生成JSP以及oSGi这类频繁自定义类加载器的场景中，通常都需要Java虚拟机具备类型卸载的能力，以保证不会对方法区造成过大的内存压力。

8. 总结



运行时数据区4-对象的实例化内存布局与访问定位+直接内存

1. 对象的实例化



1.1 创建对象的方式

- new(最常见的方式)
 - 变形1： Xxx的静态方法
 - 变形2： XxBuilder/XxoxFactory的静态方法
- Class的新Instance ()： 反射的方式，只能调用空参的构造器，权限必须是public
- Constructor的新Instance (Xxx)： 反射的方式，可以调用空参、带参的构造器，权限没有要求
- 使用clone ()： 不调用任何构造器，当前类需要实现Cloneable接口，实现clone ()
- 使用反序列化： 从文件中、从网络中获取一个对象的二进制流
- 第三方库Objenesis

1.2 创建对象的步骤

1. 判断对象对应的类是否加载、链接、初始化
2. 为对象分配内存
 1. 如果内存规整一指针碰撞
 2. 如果内存不规整：
 1. 虚拟机需要维护一个列表
 2. 空闲列表分配
3. 处理并发安全问题
 1. 采用CAS配上失败重试保证更新的原子性
 2. 每个线程预先分配一块TLAB
4. 初始化分配到的空间一所有属性设置默认值，保证对象实例字段在不赋值时可以直接使用
5. 设置对象的对象头

6. 执行init方法进行初始化

1) 判断对象对应的类是否加载、链接、初始化

虚拟机遇到一条new指令，首先去检查这个指令的参数能否在Metaspace的常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化。（即判断类元信息是否存在）。

1. 如果没有，那么在双亲委派模式下，使用当前类加载器以ClassLoader+包名+类名为Key进行查找对应的.class文件。如果没有找到文件，则抛出ClassNotFoundException异常，
2. 如果找到，则进行类加载，并生成对应的Class类对象

2) 为对象分配内存

首先计算对象占用空间大小，接着在堆中划分一块内存给新对象。如果实例成员变量是引用变量，仅分配引用变量空间即可，即4个字节大小。

1. 如果内存规整，使用指针碰撞

如果内存是规整的，那么虚拟机将采用的是指针碰撞法（BumpThePointer）来为对象分配内存。意思是所有用过的内存都在一边，空闲的内存都在另外一边，中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针向空闲那边挪动一段与对象大小相等的距离罢了。如果垃圾收集器选择的是Serial、ParNew这种基于压缩算法的，虚拟机采用这种分配方式。一般使用带有compact（整理）过程的收集器时，使用指针碰撞。

如果内存不规整，虚拟机需要维护一个列表，使用空闲列表分配

2. 如果内存不是规整的，已使用的内存和未使用的内存相互交错，那么虚拟机将采用的是空闲列表法来为对象分配内存。意思是虚拟机维护了一个列表，记录上哪些内存块是可用的，再分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的内容。这种分配方式成为“空闲列表（Free List）”。

说明：选择哪种分配方式由Java堆是否规整决定，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

给对象的属性赋值的操作：

- ① 属性的默认初始化
- ② 显式初始化
- ③ 代码块中初始化
- ④ 构造器中初始化

3) 处理并发安全问题

在分配内存空间时，另外一个问题是及时保证new对象时候的线程安全性：创建对象是非常频繁的操作，虚拟机需要解决并发问题。虚拟机采用了两种方式解决并发问题：

- CAS（Compare And Swap）失败重试、区域加锁：保证指针更新操作的原子性；

- TLAB把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲区，（TLAB， Thread Local Allocation Buffer）虚拟机是否使用 TLAB，可以通过一XX: +/−UseTLAB参数来设定。

4) 初始化分配到的空间

内存分配结束，虚拟机将分配到的内存空间都初始化为零值（不包括对象头）。这一步保证了对象的实例字段在Java代码中可以不用赋初始值就可以直接使用，程序能访问到这些字段的数据类型所对应的零值。

5) 设置对象的对象头

将对象的所属类（即类的元数据信息）、对象的HashCode和对象的GC信息、锁信息等数据存储在对象的对象头中。这个过程的具体设置方式取决于JVM实现。

6) 执行init方法进行初始化

在Java程序的视角看来，初始化才正式开始。初始化成员变量，执行实例化代码块，调用类的构造方法，并把堆内对象的首地址赋值给引用变量。因此一般来说（由字节码中是否跟随有invokespecial指令所决定），new指令之后会接着就是执行方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全创建出来。

代码示例

```

/**
 * 测试对象实例化的过程
 * ① 加载类元信息 - ② 为对象分配内存 - ③ 处理并发问题 - ④ 属性的默认初始化（零值初始化）
 * - ⑤ 设置对象头的信息 - ⑥ 属性的显式初始化、代码块中初始化、构造器中初始化
 *
 * 给对象的属性赋值的操作：
 * ① 属性的默认初始化 - ② 显式初始化 / ③ 代码块中初始化 - ④ 构造器中初始化
 *
 */
public class Customer{
    int id = 1001;
    String name;
    Account acct;

    {
        name = "匿名客户";
    }
    public Customer(){
        acct = new Account();
    }
}

class Account{
}

```

2. 对象的内存布局

对象头 (Header)

包含两部分

- 运行时元数据
 - 哈希值 (HashCode)
 - GC分代年龄
 - 锁状态标志
 - 线程持有的锁
 - 偏向线程ID
 - 偏向时间戳
- 类型指针：指向类元数据的InstanceKlass，确定该对象所属的类型

说明：如果是数组，还需记录数组的长度

实例数据 (Instance Data)

说明：它是对象真正存储的有效信息，包括程序代码中定义的各种类型的字段（包括从父类继承下来的和本身拥有的字段）规则：

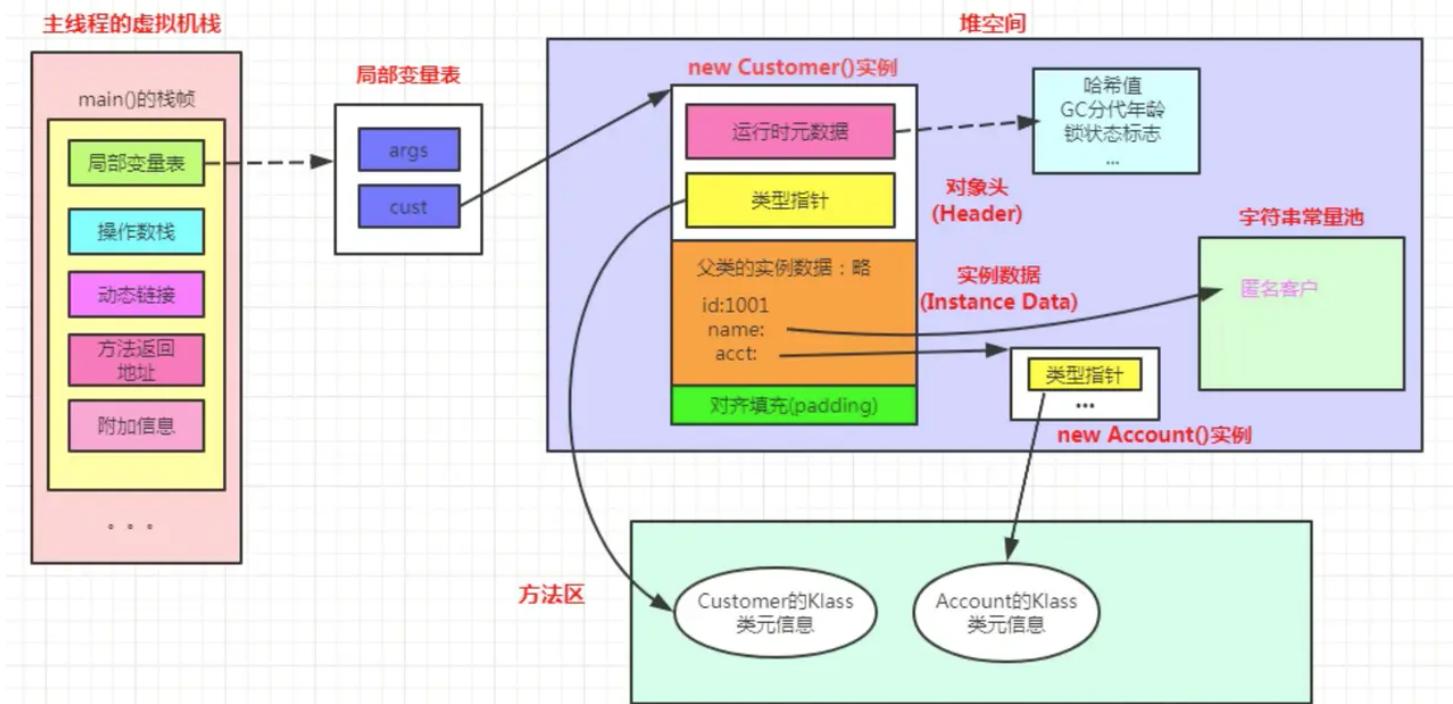
- 相同宽度的字段总被分配在一起
- 父类中定义的变量会出现在子类之前
- 如果CompactFields参数为true（默认为true），子类的窄变量可能插入到父类变量的空隙

对齐填充（Padding）

不是必须的，也没特别含义，仅仅起到占位符作用

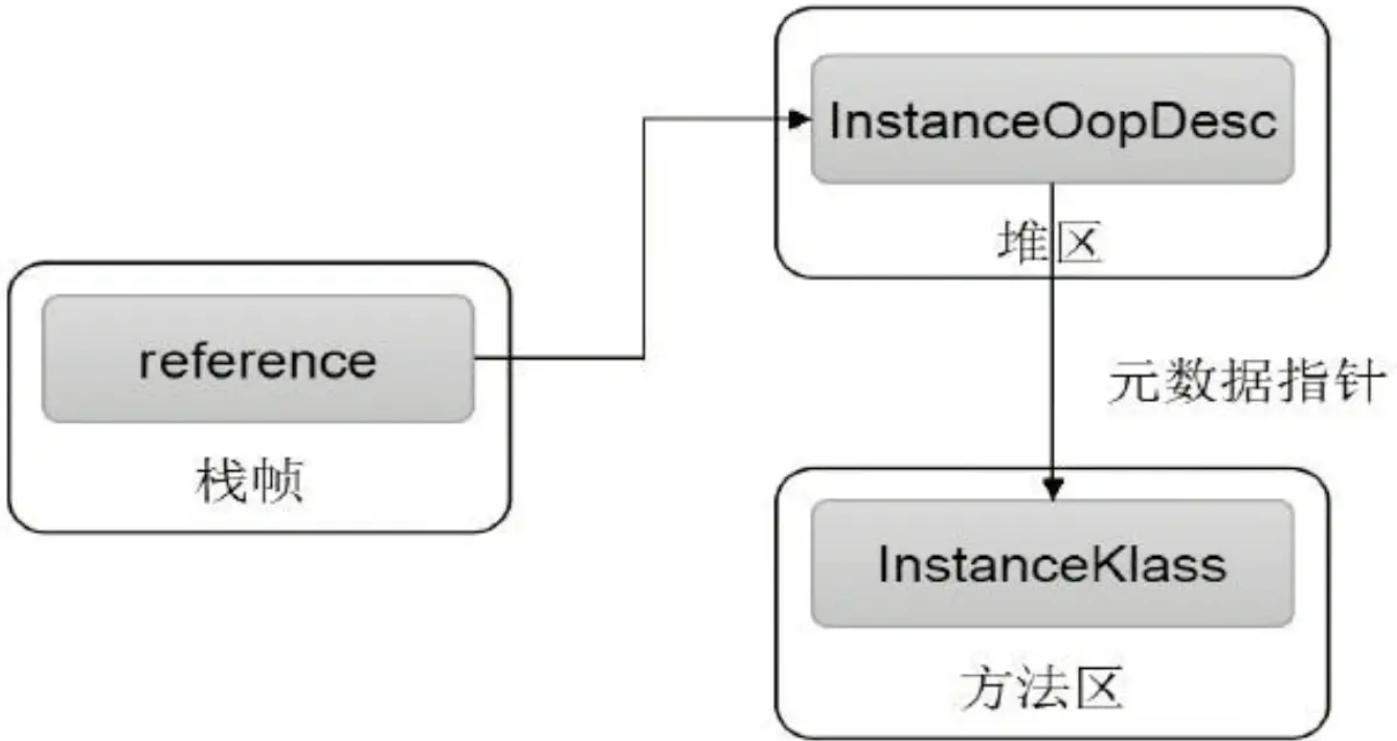
小结

```
public class CustomerTest {  
    public static void main(String[] args) {  
        Customer cust = new Customer();  
    }  
}
```



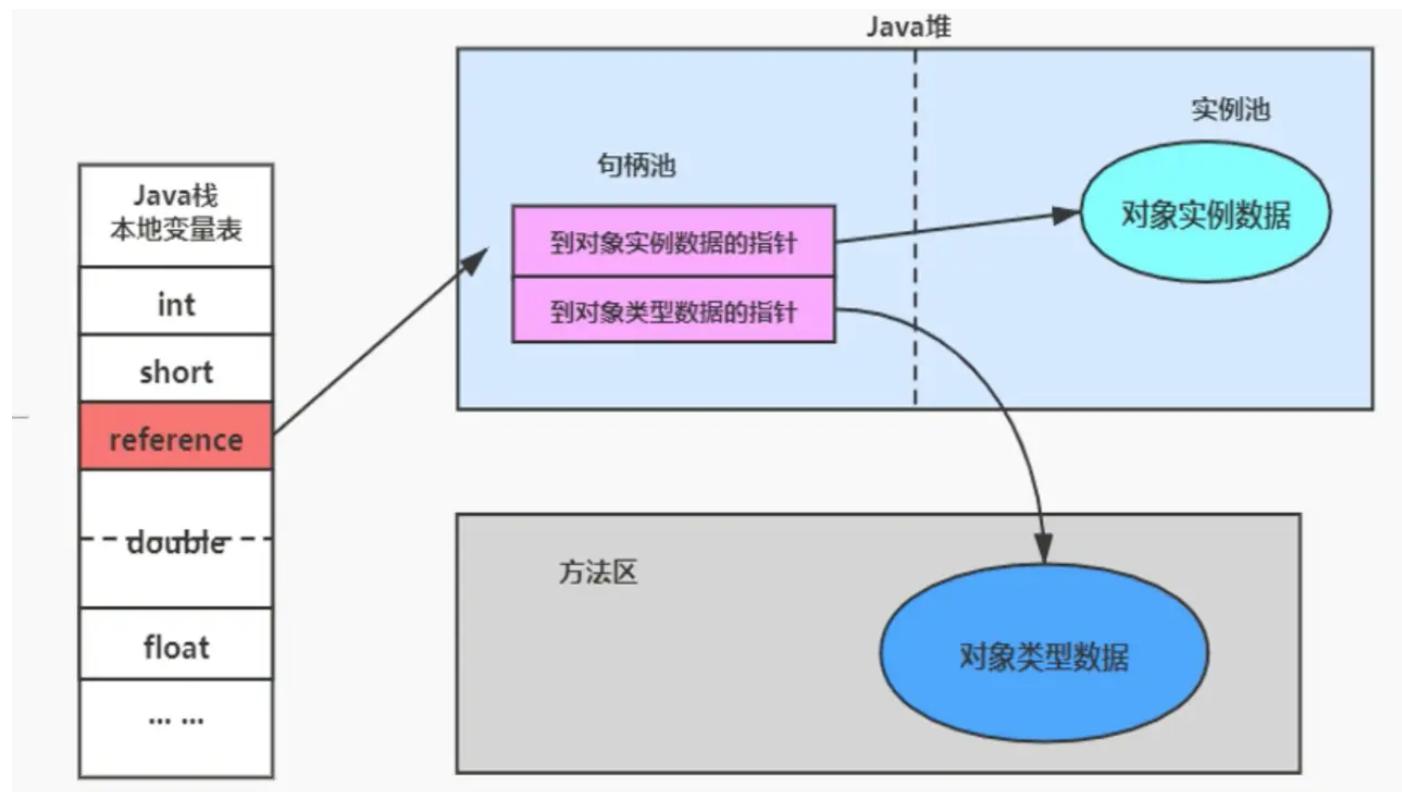
3. 对象的访问定位

JVM是如何通过栈帧中的对象引用访问到其内部的对象实例的呢？-> 定位,通过栈上reference访问

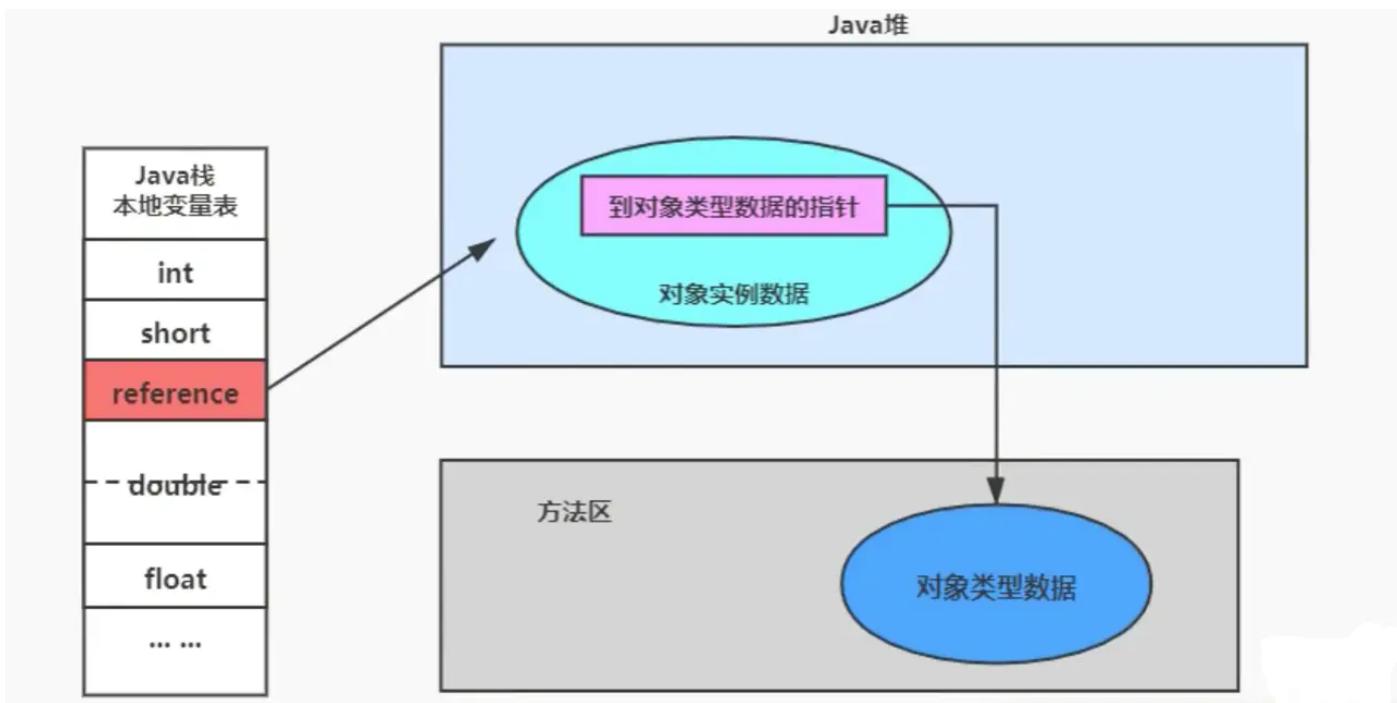


对象访问的主要方式有两种

- 句柄访问



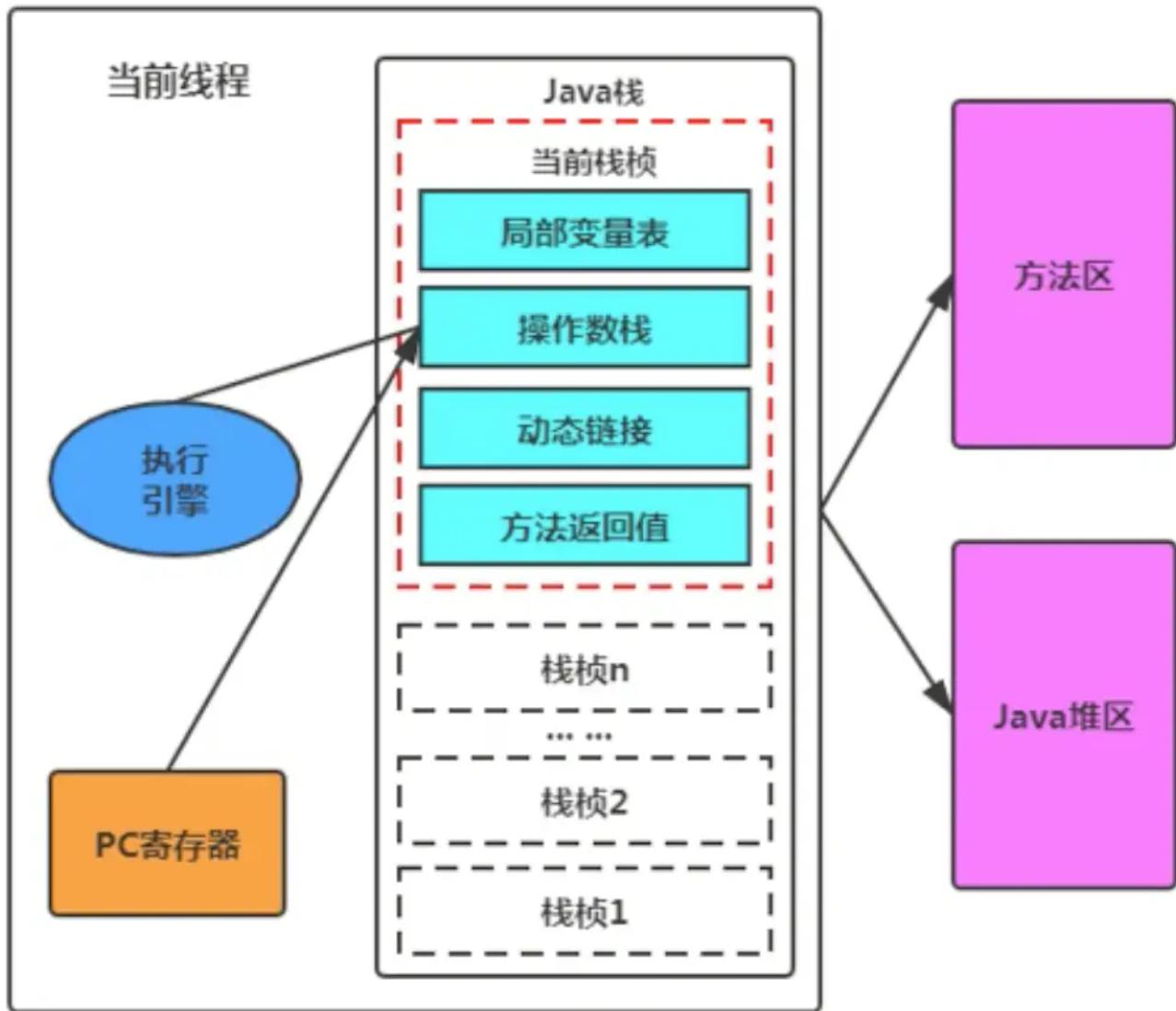
- 直接指针(HotSpot采用)



执行引擎 (Execution Engine)

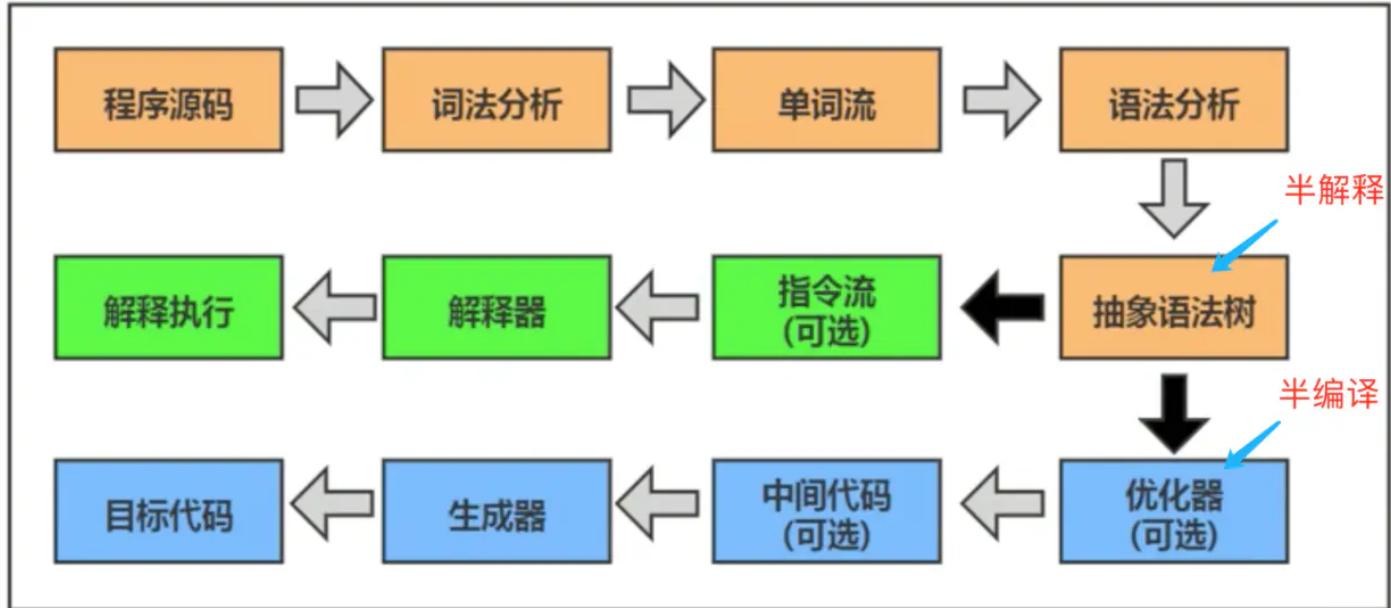
执行引擎概述

- 执行引擎是Java虚拟机的核心组成部分之一
- 虚拟机是一个相对于“物理机”的概念，这两种机器都有代码执行能力，其区别是物理机的执行引擎是直接建立在处理器、缓存、指令集和操作系统层面上的，而虚拟机的执行引擎则是由软件自行实现的，因此可以不受物理条件制约地定制指令集与执行引擎的结构体系，能够执行那些不被硬件直接支持的指令集格式。
- JVM的主要任务是负责装载字节码到其内部，但字节码并不能够直接运行在操作系统之上，因为字节码指令并非等价于本地机器指令，它内部包含的仅仅只是一些能够被JVM识别的字节码指令、符号表和其他辅助信息
- 那么，如果想让一个Java程序运行起来、执行引擎的任务就是将字节码指令解释/编译为对应平台上的本地机器指令才可以。简单来说，JVM中的执行引擎充当了将高级语言翻译为机器语言的译者。
- 执行引擎的工作过程
 - 从外观上来看，所有的Java虚拟机的执行引擎输入、输出都是一致的：输入的是字节码二进制流，处理过程是字节码解析执行的等效过程，输出的是执行结果。
 - 1) 执行引擎在执行的过程中究竟需要执行什么样的字节码指令完全依赖于PC寄存器。
 - 2) 每当执行完一项指令操作后，PC寄存器就会更新下一条需要被执行的指令地址。
 - 3) 当然方法在执行的过程中，执行引擎有可能会通过存储在局部变量表中的对象引用准确定位到存储在Java堆区中的对象实例信息，以及通过对对象头中的元数据指针定位到目标对象的类型信息。

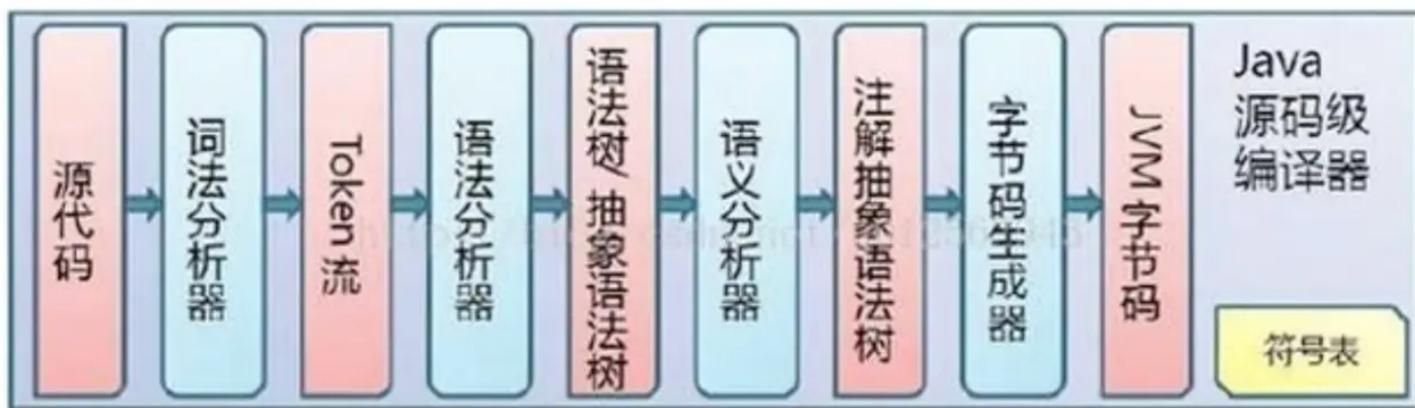


Java代码编译和执行过程

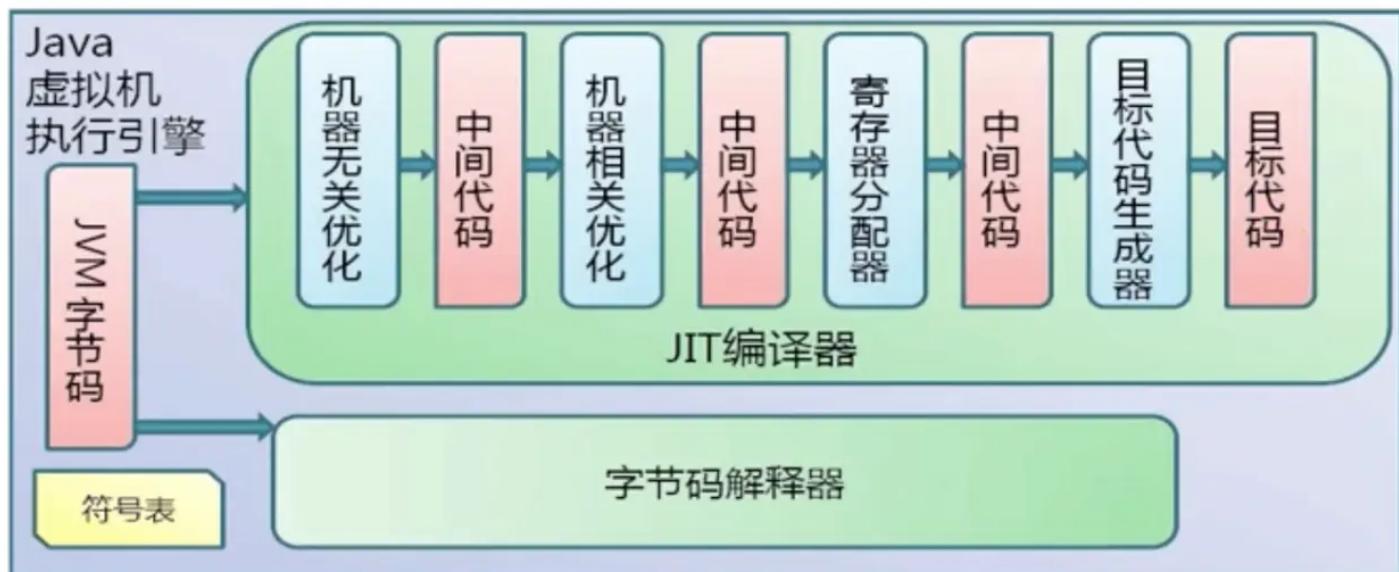
大部分的程序代码转换成物理机的目标代码或虚拟机能执行的指令集之前，都需要经过下面图中的各个步骤：



Java代码编译是由Java源码编译器来完成，流程图如下所示：



Java字节码的执行是由JVM执行引擎来完成，流程图如下所示：



什么是解释器（Interpreter），什么是JIT编译器？

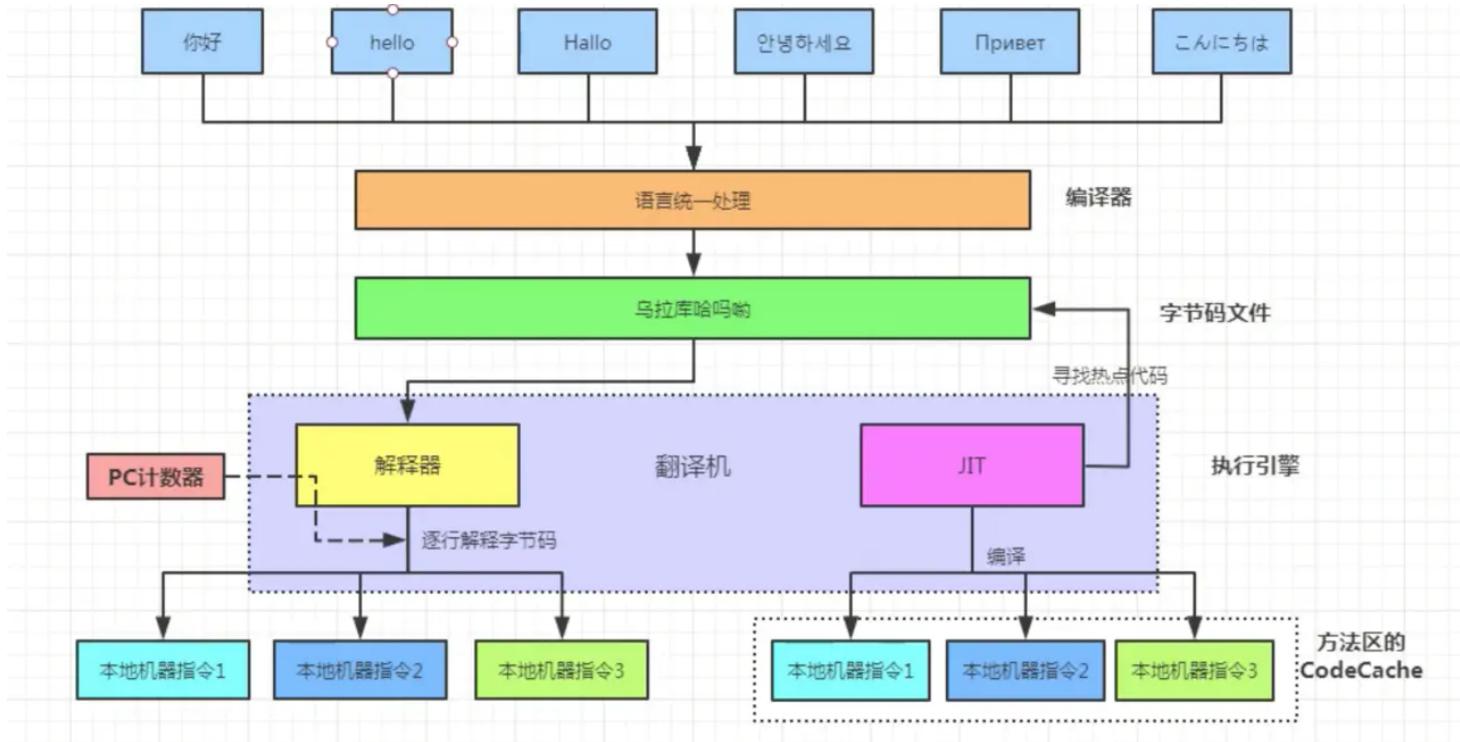
解释器：当Java虚拟机启动时会根据预定义的规范对字节码采用逐行解释的方式执行，将每条字节码文件中的内容“翻译”为对应平台的本地机器指令执行。

JIT（Just In Time Compiler）编译器（即时编译器）：就是虚拟机将源代码直接编译成和本地机器平台相关的机器语言。

为什么说Java是半编译半解释型语言？

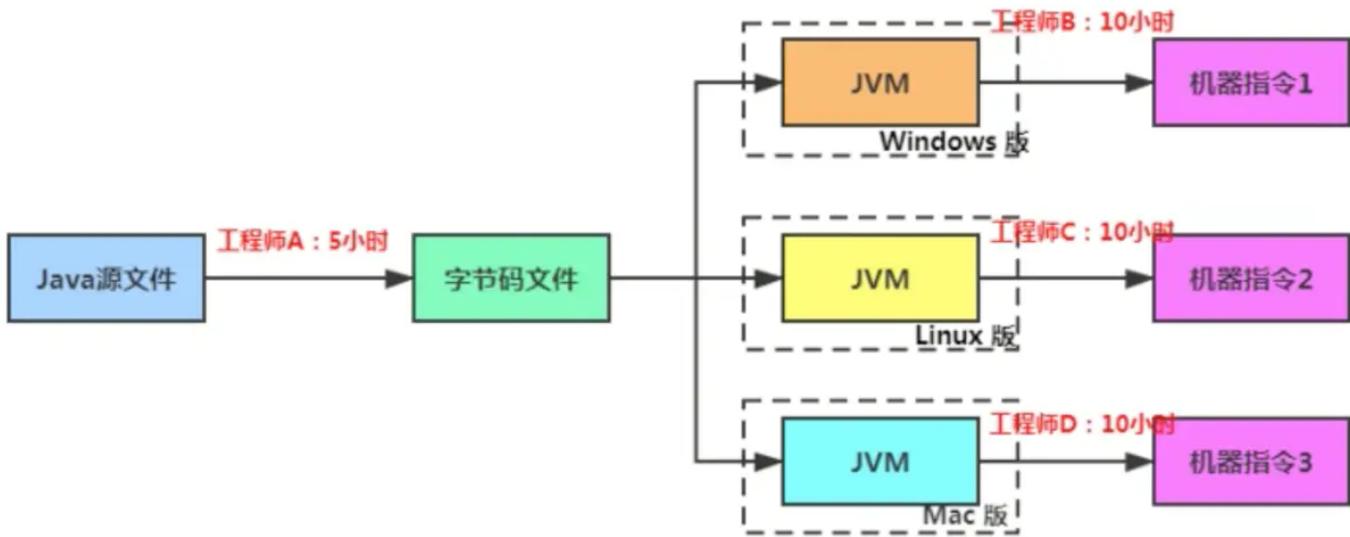
JDK1.0时代，将Java语言定位为“解释执行”还是比较准确的。再后来，Java也发展出可以直接生成本地代码的编译器。

现在JVM在执行Java代码的时候，通常都会将解释执行与编译执行二者结合起来进行。



解释器

JVM设计者们的初衷仅仅只是单纯地为了满足Java程序实现跨平台特性，因此避免采用静态编译的方式直接生成本地机器指令，从而诞生了实现解释器在运行时采用逐行解释字节码执行程序的想法。



- 解释器真正意义上所承担的角色就是一个运行时“翻译者”，将字节码文件中的内容“翻译”为对应平台的本地机器指令执行。
- 当一条字节码指令被解释执行完成后，接着再根据PC寄存器中记录的下一条需要被执行的字节码指令执行解释操作。

在Java的发展历史里，一共有两套解释执行器，即古老的字节码解释器、现在普遍使用的模板解释器。

- 字节码解释器在执行时通过纯软件代码模拟字节码的执行，效率非常低下。
- 而模板解释器将每一条字节码和一个模板函数相关联，模板函数中直接产生这条字节码执行时的机器码，从而很大程度上提高了解释器的性能。
 - 在HotSpot VM中，解释器主要由Interpreter模块和Code模块构成。
 - Interpreter模块：实现了解释器的核心功能
 - Code模块：用于管理HotSpot VM在运行时生成的本地机器指令

1. jdk1.7之前 ↵