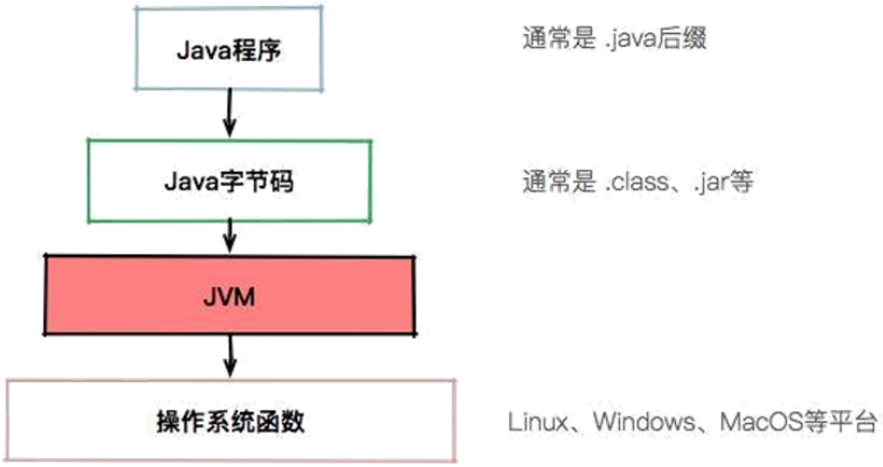
为什么需要 JVM？它处在什么位置？

JVM介绍：

JVM等同于操作系统，在系统中的位置可参考下图



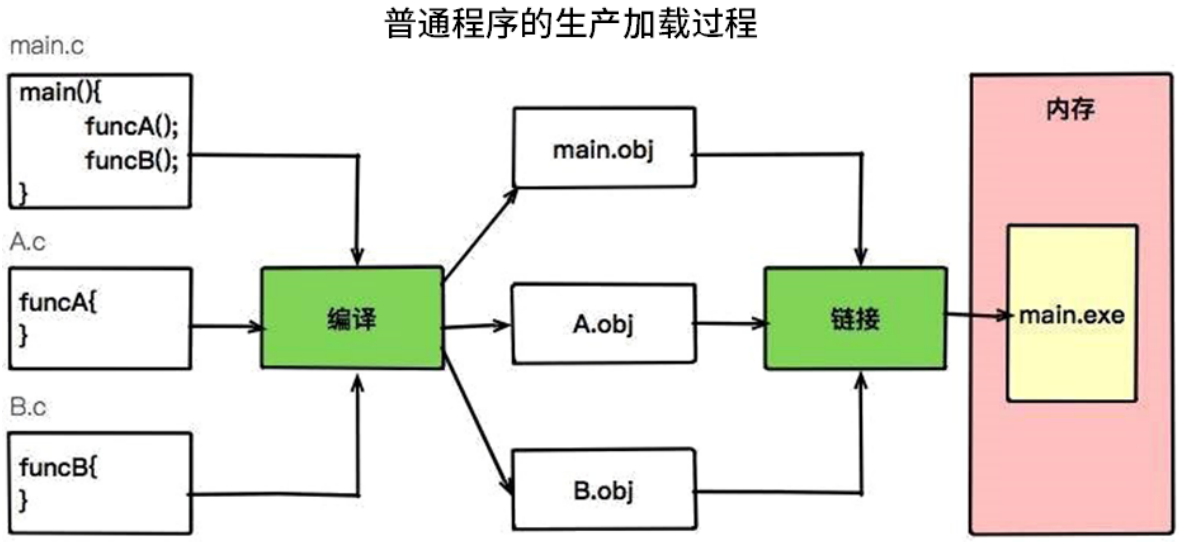
从图中可以看到，有了 JVM 这个抽象层之后，Java 就可以实现跨平台了。JVM 只需要保证能够正确执行 .class 文件，就可以运行在诸如 Linux、Windows、MacOS 等平台上了。

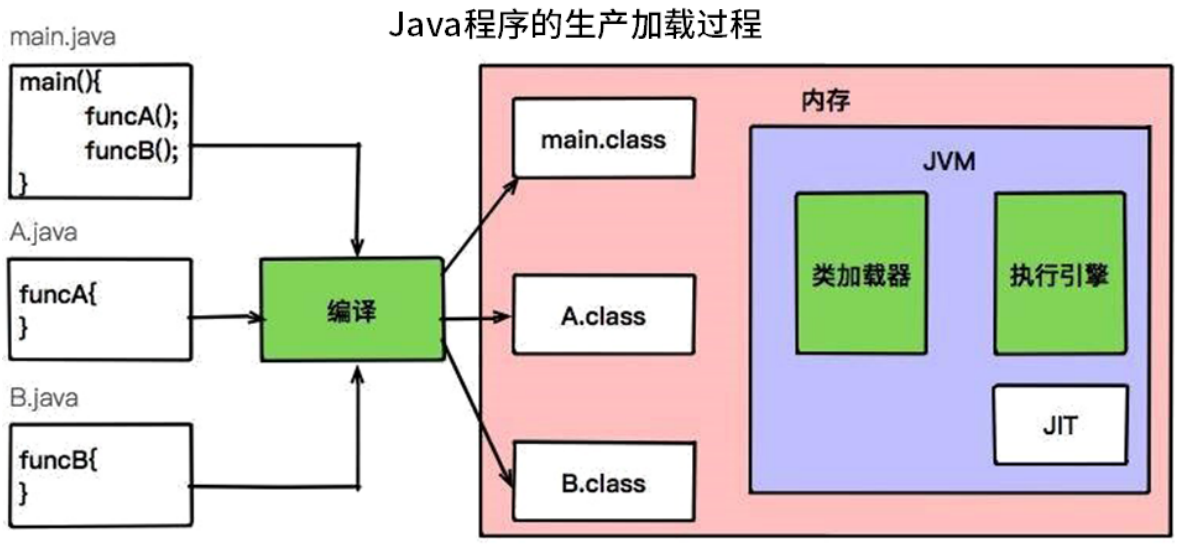
JVM 与操作系统之间的关系：

JVM 上承开发语言，下接操作系统，它的中间接口就是字节码。

Java 程序和我们通常使用的 C++ 程序有什么不同呢？

这里用两张图进行说明。

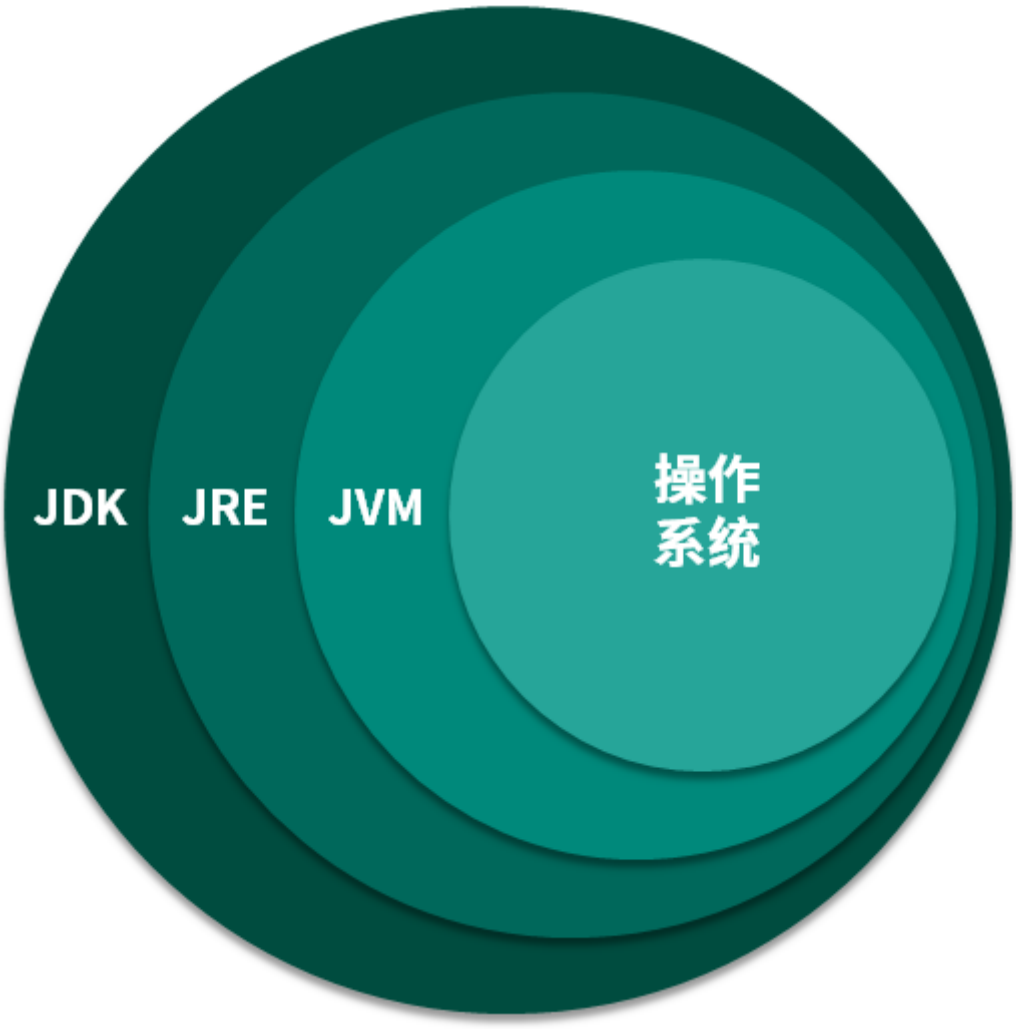




对比这两张图可以看到 C++ 程序是编译成操作系统能够识别的 .exe 文件，而 Java 程序是编译成 JVM 能够识别的 .class 文件，然后由 JVM 负责调用系统函数执行程序。

JVM、JRE、JDK的关系：

JDK>JRE>JVM

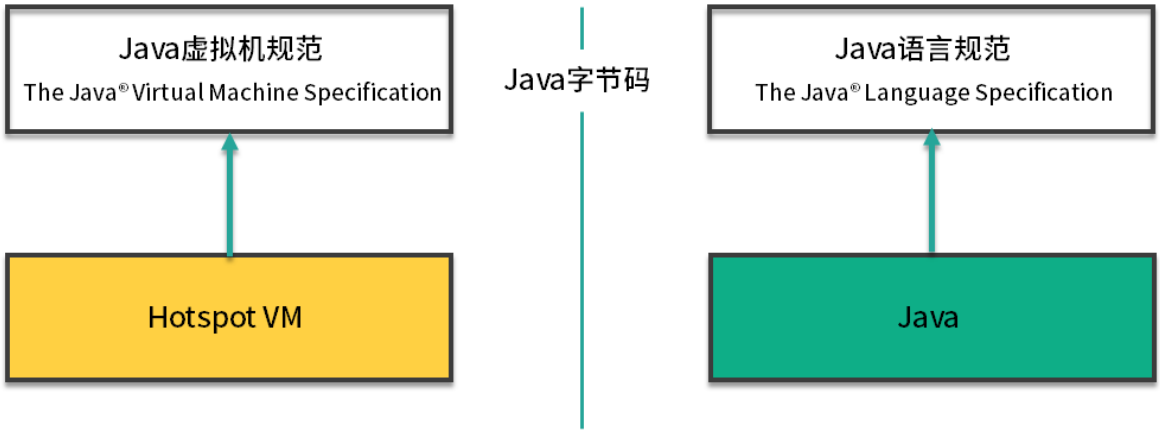


JVM：编译.class文件

JRE：JRE提供Java应用运行所需的最小支撑环境，它包括JVM、核心类、和一些支持文件，使java程序能够在浏览器运行

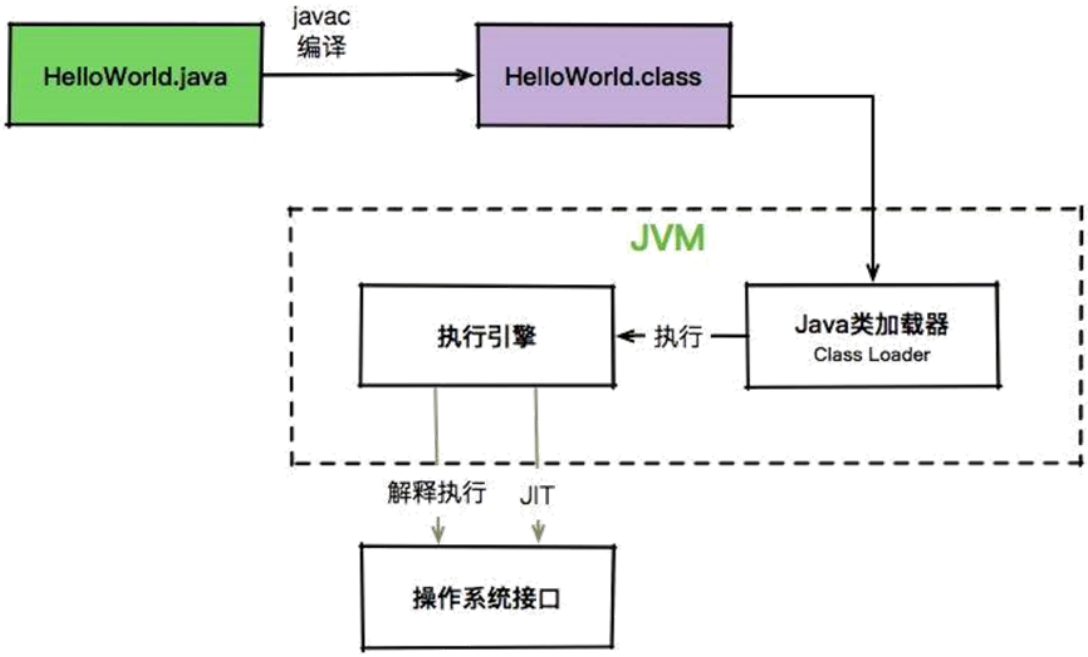
JDK：比上两层对一些小工具，比如 javac、java、jar 等。它是 Java 开发的核心，让外行也可以炼剑！

Java开发人员必须搞懂的两个规范：



左半部分是 Java 虚拟机规范，其实就是为输入和执行字节码提供一个运行环境。右半部分是我们常说的 Java 语法规范，比如 switch、for、泛型、lambda 等相关的程序，最终都会编译成字节码。而连接左右两部分的桥梁依然是 Java 的字节码。

java代码运行过程如下：

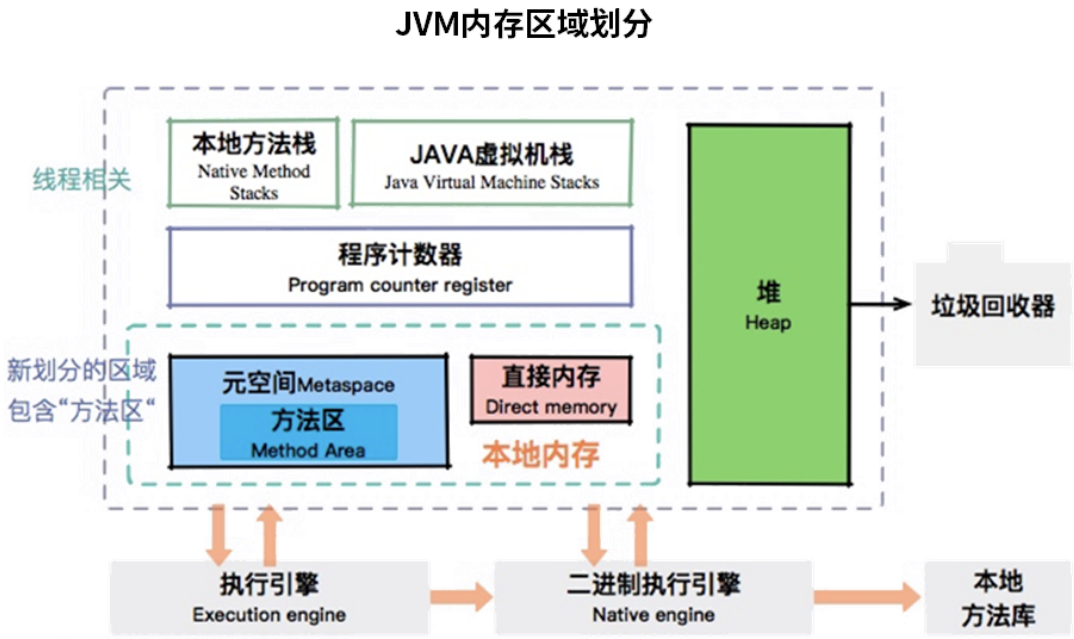


为什么java要研发系统需要JVM：

虚拟环境要解决字节加载、自动垃圾回收、并发等一系列问题。**JVM是一个规范**，定义了.class文件的结构、加载机制、数据存储、运行时栈等诸多内容，最常用的 JVM 实现就是 Hotspot。

JVM的内存区域划分

JVM的内存区域划分如图：



从图中可以看出：

- 堆：JVM 堆中的数据是共享的，是占用内存最大的一块区域。
- 执行引擎：可以执行字节码的模块
- 执行引擎在线程切换时怎么恢复？依靠的就是程序计数器。
- JVM 的内存划分与多线程是息息相关的。像我们程序中运行时用到的栈，以及本地方法栈，它们的维度都是线程。
- 本地内存包含元数据区和一些直接内存。

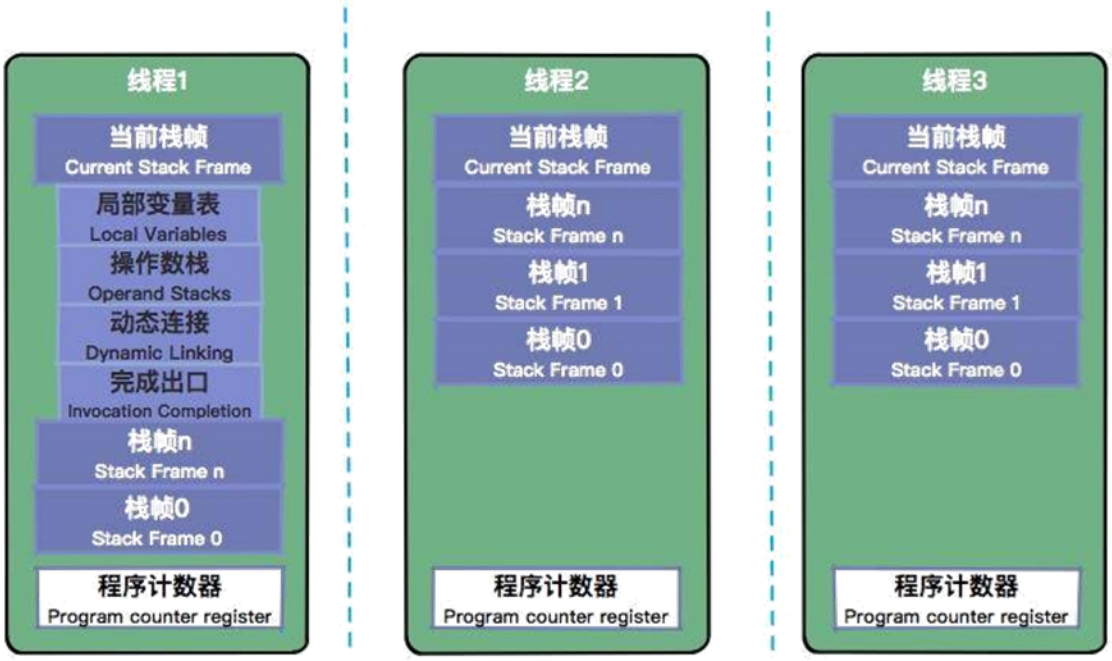
虚拟机栈:

栈的特性为先进先出，虚拟机栈是基于线程的，生命周期和线程是一样的，栈帧都出栈后，线程也就结束了。

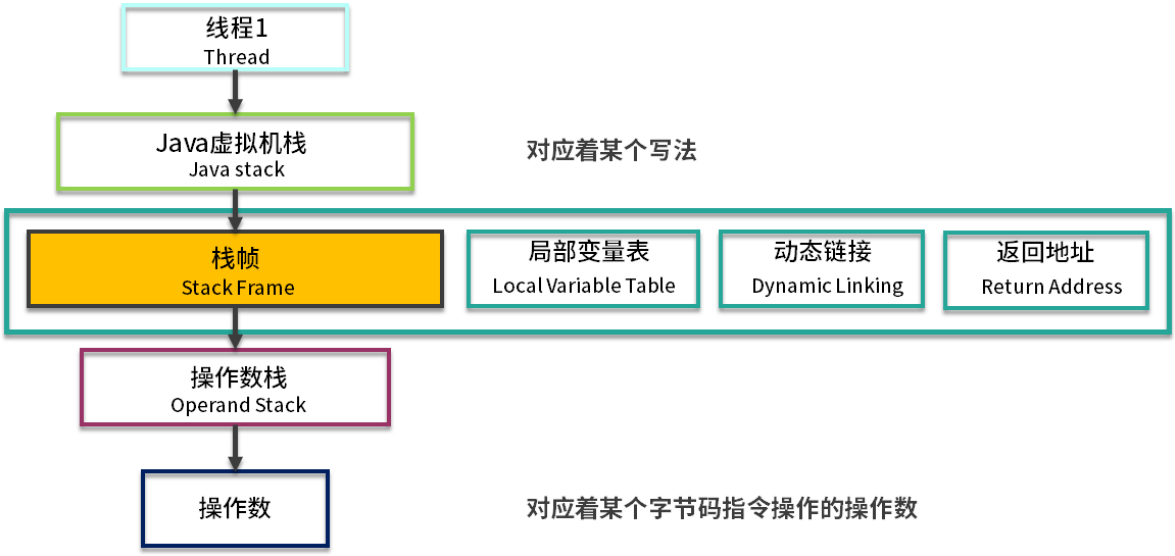
每个栈帧包含四个区域：

- 局部变量表
- 操作数栈
- 动态连接
- 返回地址

栈帧、线程、程序计数器关系如下图：



本地方法栈是和虚拟机栈非常相似的一个区域，它服务的对象是 native 方法。你甚至可以认为虚拟机栈和本地方法栈是同一个区域，这并不影响我们对 JVM 的了解。



由上图中看出：

1. 这里有一个两层的栈,第一层是栈帧，对应着方法；第二层是方法的执行，对应着操作数。
2. 所有的字节码指令，其实都会抽象成对栈的入栈出栈操作。执行引擎只需要傻瓜式的按顺序执行，就可以保证它的正确性。
3. 返回地址（ReturnAdress）就是指向特定指令内存地址的指针。

程序计数器：

唯一不会出现OOM（内存溢出）的内存区域，线程之间切换时，知道线程已经执行到什么地方了，以便继续执行，不用重新开始，每个线程都有一个自己的程序计数器。

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=6, args_size=1
       0: new           #3                  // class A
       3: dup
       4: invokespecial #4                  // Method "<init>":()V
       7: astore_1
       8: ldc2_w        #5                  // long 654321
      11: lstore_2
      12: aload_1
      13: lload_2
      14: invokevirtual #7                  // Method fig:(J)J
      17: lstore       4
      19: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      22: lload         4
      24: invokevirtual #9                  // Method java/io/PrintStream.println:(J)V
      27: return
    LineNumberTable:
      line 9: 0
      line 10: 8
      line 12: 12
      line 14: 19
      line 15: 27
```

如上图，每个 opcode 前面，都有一个序号(红框中的偏移地址)，你可以认为它们是程序计数器的内容。

堆:

堆是 JVM 上最大的内存区域，我们申请的几乎所有的对象，都是在这里存储的。我们常说的垃圾回收，操作的对象就是堆,。

堆空间一般是程序启动时，就申请了，但是并不一定会全部使用。

堆内存 = 年轻代 + 年老代 + 永久代

一个对象创建的时候，到底是在堆上分配，还是在栈上分配取决于两个方面：

- 对象的类型：

  Java 的对象可以分为基本数据类型和普通对象

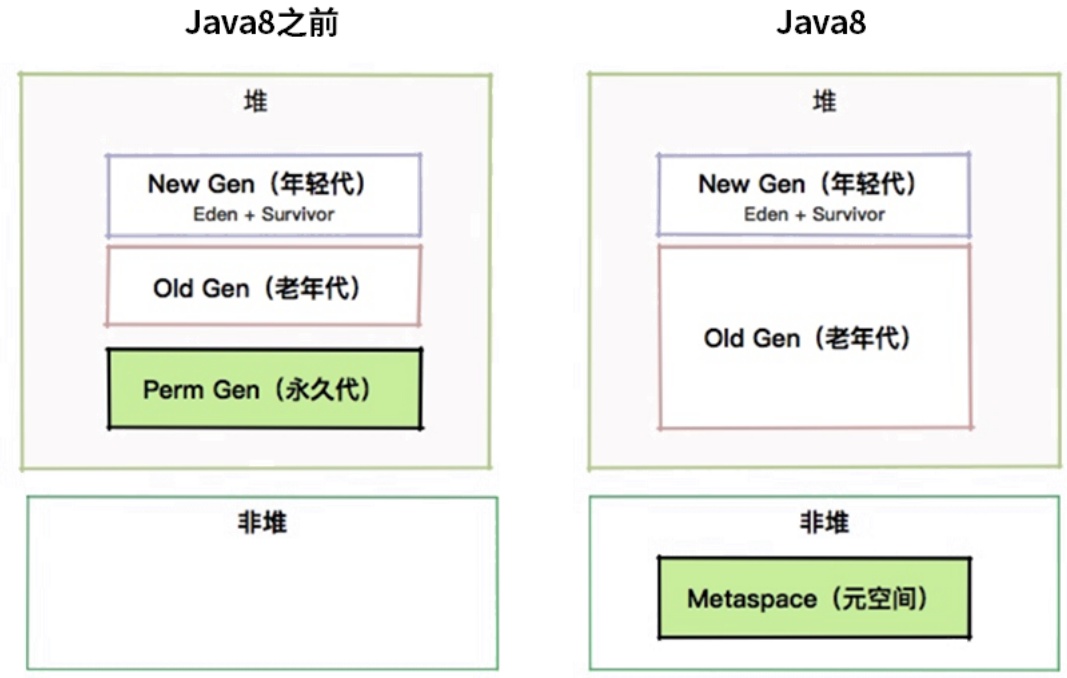
  - 基本数据类型（byte、short、int、long、float、double、char)：
    - 当你在方法体内声明了基本数据类型的对象，它就会在栈上直接分配。
    - 除了上面的情况，都是在堆上分配。

      > 注：像 int[] 数组这样的内容，是在堆上分配的。数组并不是基本数据类型。

  - 普通对象：
    - 首先在堆上创建对象，其他地方使用的其实是它的引用。
    - 需要引用对象时保存在虚拟机栈的局部变量表中
- 在 Java 类中存在的位置

堆是所有线程共享的，如果是多个线程访问，会涉及数据同步问题。这同样是个大话题，我们在这里先留下一个悬念。

元空间:



想一下类与对象的区别：对象是一个活生生的个体，可以参与到程序的运行中；类更像是一个模版，定义了一系列属性和操作。

然后，元空间的好处也是它的坏处。使用非堆可以使用操作系统的内存，JVM 不会再出现方法区的内存溢出（这也是永久代移除的原因）；但是，无限制的使用会造成操作系统的死亡。所以，一般也会使用参数 -XX:MaxMetaspaceSize 来控制大小。

方法区，作为一个概念，依然存在。它的物理存储的容器，就是 Metaspace。我们将在后面的课时中，再次遇到它。现在，你只需要了解到，这个区域存储的内容，包括：类的信息、常量池、方法数据、方法代码就可以了。

元空间和永久代：

在Java1.8中，HotSpot虚拟机已经将方法区(永久代)从堆中移除，取而代之的就是元空间，元空间放入本地内存。

永久代和元空间的作用都是存储类的元数据，用来存储class相关信息，包括class对象的Method，Field等。

为什么要废除永久代?

1、现实使用中易出问题。

- 由于永久代内存经常不够用或者发生内存泄露，爆出异常 java.lang.OutOfMemoryError: PermGen 。
- 类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。

2、永久代会为GC带来不必要的复杂度，而且回收效率偏低。

常量池：

常量池分为两种形态：

静态常量池：即*.class文件中的常量池，存放两大类常量：**字面量**和**符号引用。**

1. **字面量：**包含如字符串，声明为final的常量值、八大基本数据类型其中六个等。

   Byte、Short、Integer、Long、Character、Boolean 实现了常量池技术。前四种常量值范围为[-128, 127]，Character 范围为[0, 127]，Boolean 为 true 或 false，Float、Double 没有实现常量池技术。

String的范围为2^16-1=65535

2. **符号引用量**：包括类、方法、类里面字段（比如自己int a；a就是保存在常量池）的名称和描述符。

运行时常量池：

JVM完成类装载将*.class文件的常量池载入内存中，保存在**方法区**，这也是我们常说的常量池。

运行时的常量池不是固定不变的，可以通过intern方法加入。

不同版本jdk常量池的位置变化：

在JDK1.6及之前运行时常量池逻辑包含字符串常量池存放在**方法区**, 此时hotspot虚拟机对方法区的实现为永久代（位于堆内存中）

在JDK1.7 字符串常量池被从方法区拿到了**堆**中, 这里没有提到运行时常量池,也就是说字符串常量池被单独拿到堆,运行时常量池剩下的东西还在方法区, 也就是hotspot中的永久代

在JDK1.8 hotspot移除了方法区用元空间取而代之, 这时候字符串常量池还在堆, 运行时常量池还在方法区, 只不过方法区的实现从永久代变成了元空间(堆外内存)

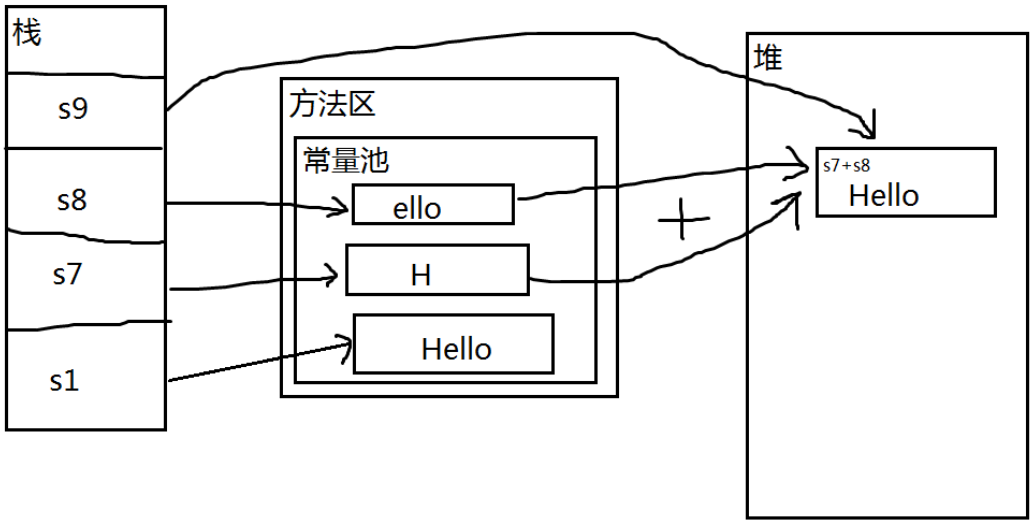
String类的intern()方法和一些面试题代码：

```
String s1 = "Hello";
String s2 = new String("Hello");
String s3 = s5.intern();
System.out.println(s1 == s3);  // true

String s5 = "Hel" + "lo";
System.out.println(s1 == s5);  //true

String s6 = "H";
String s7 = "ello";
String s8 = s6 + s7;
System.out.println(s1 == s8);  //false
```

s1 = = s6 为啥s1 和 s6地址相等呢？ 归功于intern方法，这个方法查找在常量池中是否存在一份equal相等的字符串,如果有则返回该字符串的引用,如果没有则添加自己的字符串进入常量池。（因为使用的是equal，s5就是直接拿了和他内容相等的s1来引用，当然会是true）

s5在编译期间，这种拼接会被优化，编译器直接帮你拼好，因此String s5 = "Hel" + "lo";在class文件中被优化成String s5 = "Hello"，所以s1 == s3成立。

s1 == s8也不相等，道理差不多，虽然s6、s7在赋值的时候使用的字符串字面量，但是拼接成s8的时候，s6、s7作为两个变量，都是不可预料的，编译器毕竟是编译器，不可能当解释器用，不能在编译期被确定，所以不做优化，只能等到运行时，在堆中创建s6、s7拼接成的新字符串，在堆中地址不确定，不可能与方法区常量池中的s1地址相同。



常量池的好处：

为了避免频繁的创建和销毁对象而影响系统性能，其实现了对象的共享。例如字符串常量池，在编译阶段就把所有的字符串文字放到一个常量池中。

（1）节省内存空间：常量池中所有相同的字符串常量被合并，只占用一个空间。（意思是Hello值的字符串在常量池都存在一个地址上）
（2）节省运行时间：比较字符串时，==**比equals()快。**对于两个引用变量，只用==判断引用是否相等，也就可以判断实际值是否相等。

另外一个面试题：

```
Integer a=127;
Integer b=127;
System.out.println(a==b);
Integer f=666;
Integer g=666;
System.out.println(f==g);
//结果：true flase
```

出现一个true和一个flase的原因是整型在常量池中应该是有数值限制，127在范围内，而666在范围外，所以127可以在常量池内创建，而666则是在堆内创建新对象

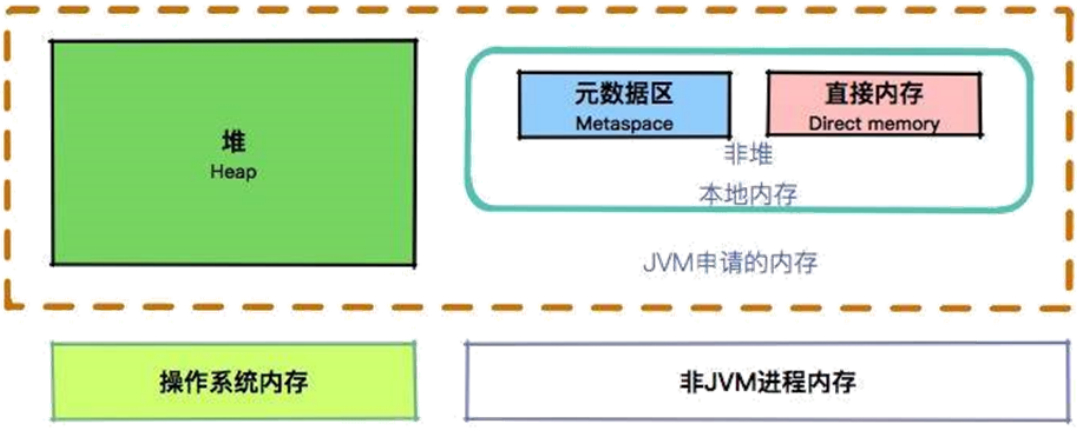
JVM内存高频面试题：

- 我们常说的字符串常量，存放在哪呢？

由于常量池，在 Java 7 之后，放到了堆中，我们创建的字符串，将会在堆上分配。

- 堆、非堆、本地内存，有什么关系？

关于它们的关系，我们可以看一张图。在我的感觉里，堆是软绵绵的，松散而有弹性；而非堆是冰冷生硬的，内存非常紧凑。



大家都知道，JVM 在运行时，会从操作系统申请大块的堆内内存，进行数据的存储。但是，堆外内存也就是申请后操作系统剩余的内存，也会有部分受到 JVM 的控制。比较典型的就是一些 native 关键词修饰的方法，以及对内存的申请和处理。

在 Linux 机器上，使用 top 或者 ps 命令，在大多数情况下，能够看到 RSS 段（实际的内存占用），是大于给 JVM 分配的堆内存的。

如果你申请了一台系统内存为 2GB 的主机，可能 JVM 能用的就只有 1GB，这便是一个限制。

总结：

JVM 的运行时区域是栈，而存储区域是堆。很多变量，其实在编译期就已经固定了。

从覆盖 JDK 的类开始掌握类的加载机制

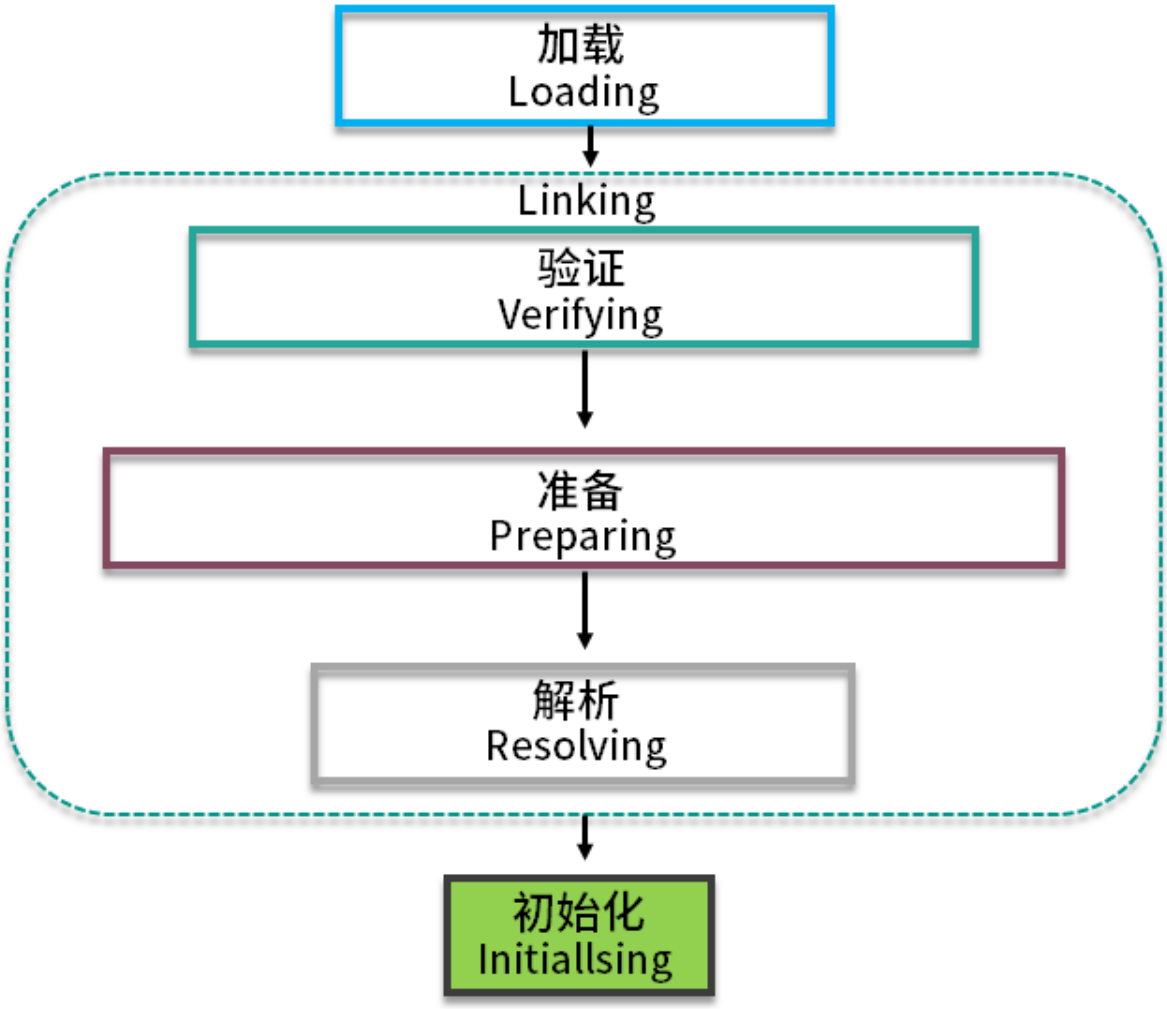
JVM 的类加载机制和 Java 的类加载机制类似，但 JVM 的类加载过程稍有些复杂。

类加载过程：

类加载器作用：

就是根据指定全限定名称将class文件加载到JVM内存，转为Class对象

现实中并不是说，我把一个文件修改成 .class 后缀，就能够被 JVM 识别。类的加载过程非常复杂，主要有这几个过程：加载、验证、准备、解析、初始化。



1.加载：

加载的主要作用是将外部的 .class 文件，加载到 Java 的方法区内。

2.验证：

不能任何 .class 文件都能加载，这样不安全，验证阶段在虚拟机整个类加载过程中占了很大一部分，不符合规范的将抛出 java.lang.VerifyError 错误。像一些低版本的 JVM，是无法加载一些高版本的类库的，就是在这个阶段完成的。

3.准备：

将为一些类变量分配内存，并将其初始化为默认值。此时，实例对象还没有分配内存，所以这些动作是在方法区上进行的。

```
public class A {
        static int a ;//类变量
        public static void main(String[] args) {
            System.out.println(a);//编译成功
        }
    }
public class A {
    public static void main(String[] args) {
        int a ;//局部变量，编译报错，没有初始值
        System.out.println(a);
    }
}
```

静态变量有两次赋初始值的过程，一次在准备阶段，赋予初始值（也可以是指定值）；另外一次在初始化阶段，赋予程序员定义的值。

局部变量就不一样了，准备阶段没有给它赋初始值，是不能使用的。

4.解析：

解析将符号引用变为直接引用。符号引用是一种定义，直接引用的对象都是存在内存的，它是指向目标的指针、相对偏移量。

有哪些解析：

- 类或接口的解析
- 类方法解析
- 接口方法解析
- 字段解析

5.初始化：

如果前面的流程一切顺利的话，接下来该初始化成员变量了，到了这一步，才真正开始执行一些字节码。

先看看面试题：

```
public class A {
     static int a = 0 ;
     static {
         a = 1;
         b = 1;
     }
     static int b = 0;

     public static void main(String[] args) {
         System.out.println(a);
         System.out.println(b);
     }
 }
```

输出结果是1 0。这就引出一个规则：static 语句块，只能访问到定义在 static 语句块之前的变量。所以下面的代码是无法通过编译的。

```
static {
         b = b + 1;//假如变为b=1可以编译并输出结果为0
 }
 static int b = 0;
```

与的区别：

总结：为类加载，为对象初始化，类加载初始化就会执行不会重复执行。对象初始化在每次新建对象都会执行一次，且初始化比类加载要晚（看下面的代码打印结果就知道了）。

先看看另一个面试题：

```
public class A {
     static {
         System.out.println("1");
```

```
    }
    public A(){
        System.out.println("2");
        }
}

public class B extends A {
    static{
        System.out.println("a");
       }
    public B(){
        System.out.println("b");
       }

    public static void main(String[] args){
        A ab = new B();
        ab = new B();
    }
 }
```

运行结果：

```
1
a
2
b
2
b
```

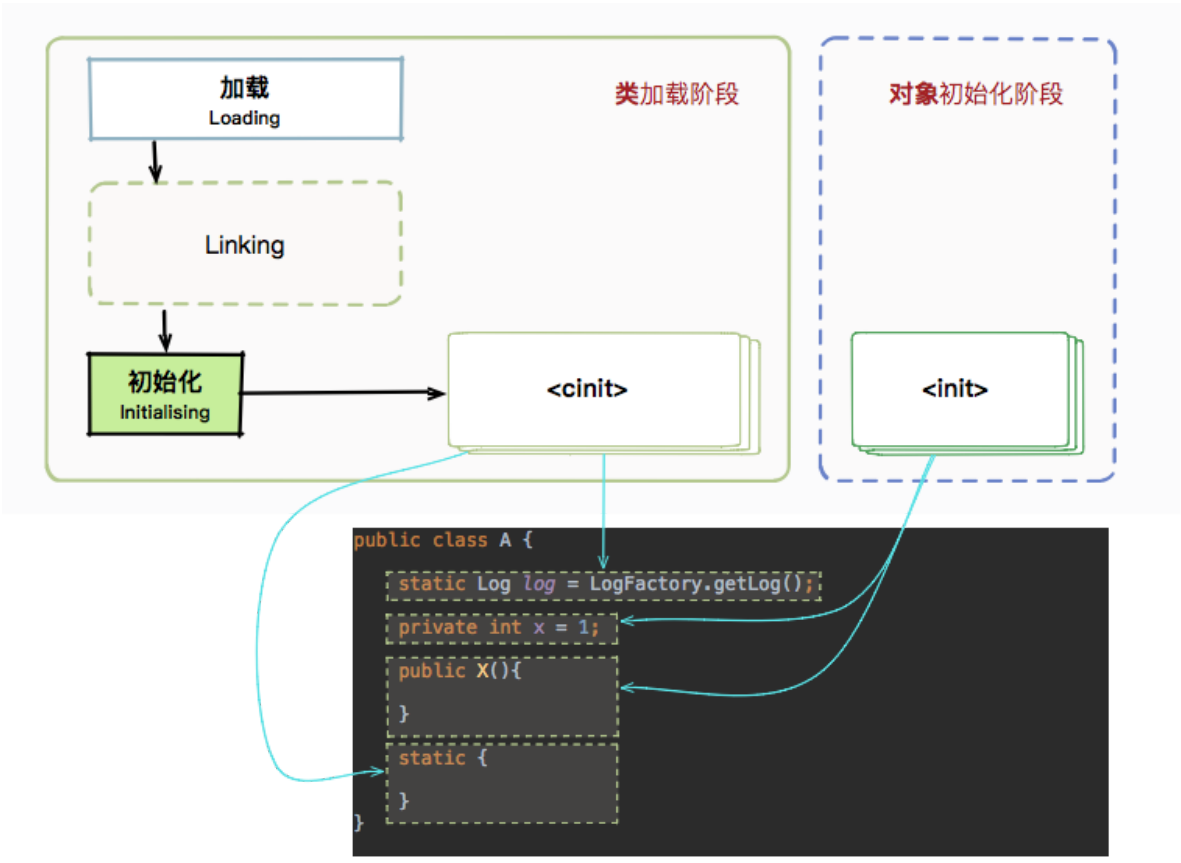
你可以看下这张图。其中 static 字段和 static 代码块，是属于类的，在类的加载的初始化阶段就已经被执行。类信息会被存放在方法区，在同一个类加载器下，这些信息有一份就够了，所以上面的 static 代码块只会执行一次，它对应的是 方法。（上面代码看出，会先运行父类再运行子类）

而对象初始化就不一样了。通常，我们在 new 一个新对象的时候，都会调用它的构造方法，就是 ，用来初始化对象的属性。每次新建对象的时候，都会执行。

所以，上面代码的 **static 代码块只会执行一次，对象的构造方法执行两次。**再加上继承关系的先后原则，不难分析出正确结果。（由结果和下图看出，static 块在类加载阶段就打印出来了）

```java
public class a {
    static int a = 0 ;
    static {
        System.out.println(1);
    }
    public static void main(String[] args) {
        int b=2;
        System.out.println(b);
        System.out.println(a);
    }

}
```

```
a › main()
ebug:  a ×
Watches  Debugger  Console
D:\software\java\jdk-11.0.5\bin\java.exe -agentlib:jdwp
Connected to the target VM, address: '127.0.0.1:60833'
1
```



类加载器：

类加载器做的就是上面 5 个步骤的事。

如果你在项目代码里，写一个 java.lang 的包，然后改写 String 类的一些行为，编译后，发现并不能生效。JRE 的类当然不能轻易被覆盖，否则会被别有用心的人利用，这就太危险了。

那类加载器是如何保证这个过程的安全性呢？其实，它是有着严格的等级制度的。

我们介绍几个不同等级的类加载器：

- **Bootstrap ClassLoader**

  加载器中的大 Boss，任何类的加载行为，都要经它过问。它的作用是加载核心类库，也就是 rt.jar、resources.jar、charsets.jar 等。当然这些 jar 包的路径是可以指定的，-Xbootclasspath 参数可以完成指定操作。这个加载器是 C++ 编写的，随着 JVM 启动。

- **Extention ClassLoader**

  扩展类加载器，主要用于加载 lib/ext 目录下的 jar 包和 .class 文件。同样的，通过系统变量 java.ext.dirs 可以指定这个目录。

  这个加载器是个 Java 类，继承自 URLClassLoader。

- **Application ClassLoader**

  这是我们写的 Java 类的默认加载器，有时候也叫作 System ClassLoader。一般用来加载 classpath 下的其他所有 jar 包和 .class 文件，我们写的代码，会首先尝试使用这个类加载器进行加载。

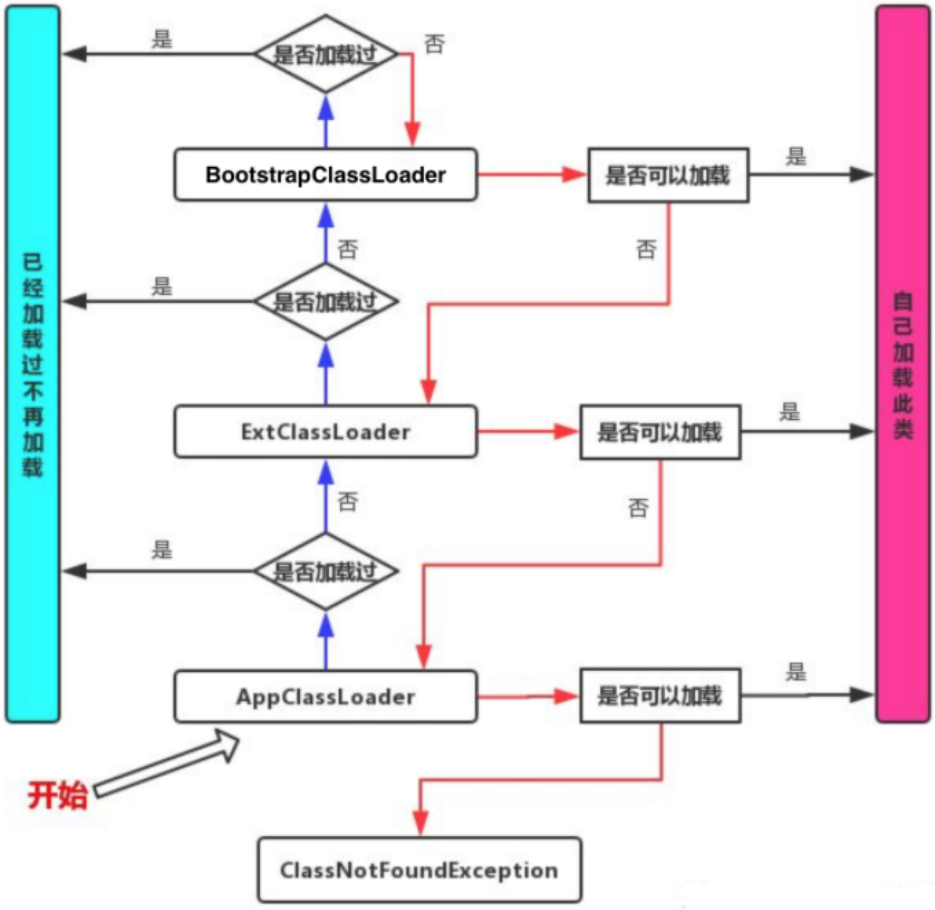
- **Custom ClassLoader**

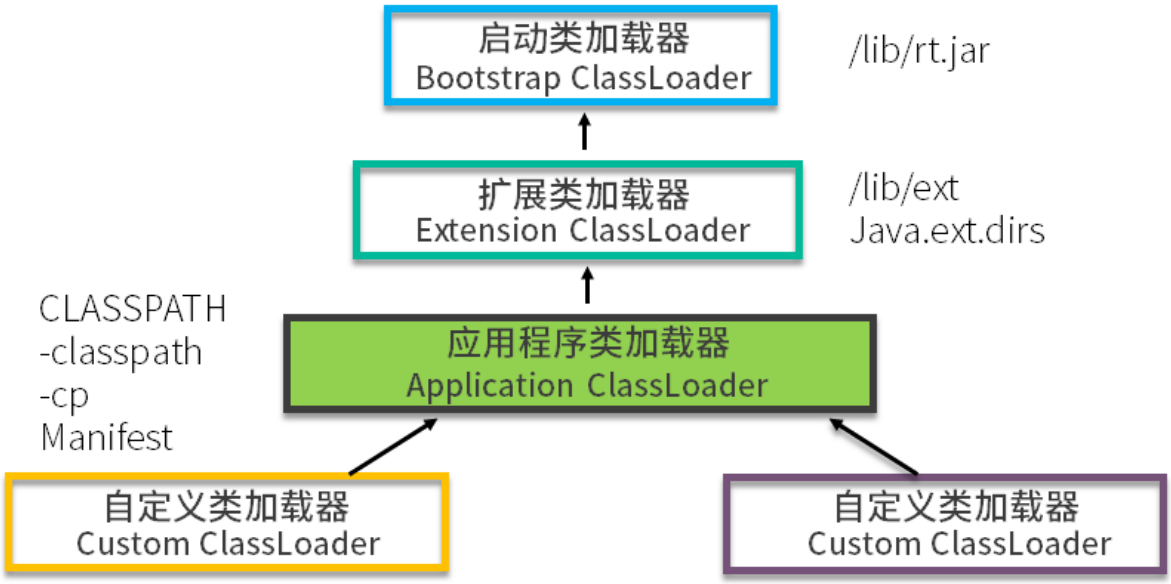
  自定义加载器，支持一些个性化的扩展功能。

双亲委派机制：

双亲委派机制介绍：

就是该类加载器加载之前，检查是否加载过，如果有那就无需再加载了。如果没有，那么会拿到**父加载器**，一层层向上类推验证，如果父加载器没有加载，会下沉到子加载器确认是否可以加载。一直到最底层，如果没有任何加载器能加载，就会抛出**ClassNotFoundException**。（至于为什么叫双亲只是翻译问题，不用太在意）





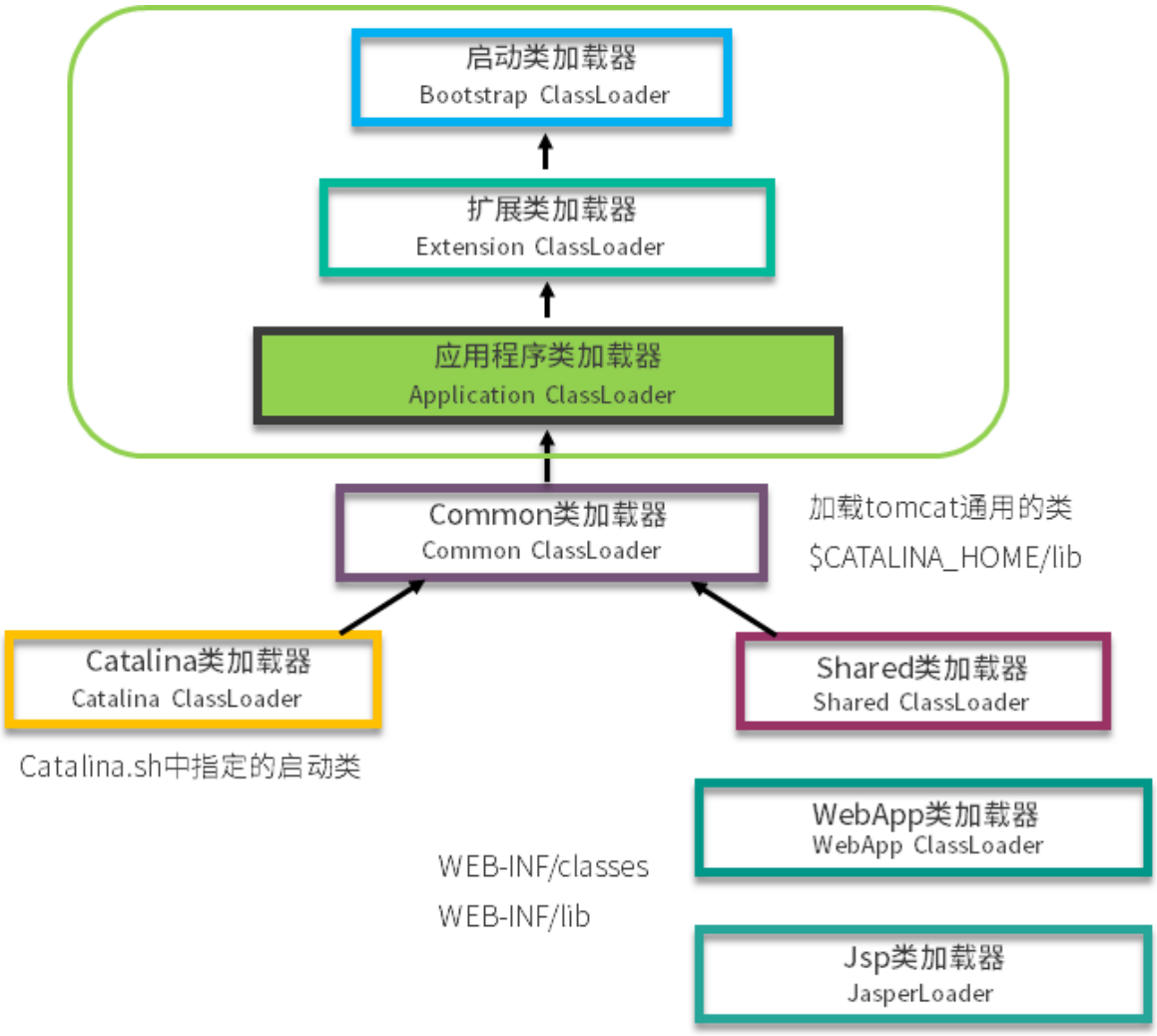
双亲委派机制有什么好处：

这种设计有个好处是，如果有人想替换系统级别的类：String.java。篡改它的实现，在这种机制下这些系统的类已经被Bootstrap classLoader加载过了（为什么？因为当一个类需要加载的时候，最先去尝试加载的就是BootstrapClassLoader），所以其他类加载器并没有机会再去加载，从一定程度上防止了危险代码的植入。

一些自定义的加载器：

1.tomcat：

tomcat的加载器层次结构：



如上图，tomcat在加载非基础类（基础类指的是像String、ArrayList）时，会打破双亲委派机制，优先使用WebAppClassLoader加载器加载，目的是为了一个JVM运行不兼容的两个版本，它们相互不影响。

那么 tomcat 是怎么打破双亲委派机制的呢？可以看图中的 WebAppClassLoader，它加载自己目录（classes、lib）下的 .class 文件，并不会传递给父类的加载器。但是，它却可以使用 SharedClassLoader 所加载的类，实现了共享和分离的功能。

> 但是你自己写一个 ArrayList，放在应用目录里，tomcat 依然不会加载。它只是自定义的加载器顺序不同，但对于顶层来说，还是一样的。

2.SPI：

以加载com.mysql.jdbc.Driver为例：

DriverManager通过Bootstrap ClassLoader加载，而com.mysql.jdbc.Driver要通过Application ClassLoader加载，为什么可以拿到Application ClassLoader加载进来的com.mysql.jdbc.Driver？

想拿到必须破坏双亲委派。

DriverManager是如何拿到com.mysql.jdbc.Driver类的：

通过将Application ClassLoader设置为线程上下文加载器实现的，在DriverManager类里通过Thread.currentThread().getContextClassLoader()拿到Application ClassLoader进行加载。

SPI加载类有什么优点：

SPI的这种加载方式，就只需要定义好接口，引入符合规范的jar包，就可以获取到实现该接口的类了。

有点类似于IOC，上层只需要指定一个配置文件路径，在初始化的时候去扫描所有符合的配置文件路径，并解析其中的内容，再去加载对应的类，就可以拿到所有该接口的实现了。

3.OSGi：

已经很少用，了解即可，它实现了模块化，每个模块可以独立安装、启动、停止、卸载。

类加载面试题：

我们能够通过一定的手段，覆盖 HashMap 类的实现么？ 来个案例试试？？？

答：可以，需要将自己的 HashMap 类，打包成一个 jar 包，然后放到 -Djava.endorsed.dirs 指定的目录中。注意类名和包名，应该和 JDK 自带的是一样的。但是，java.lang 包下面的类除外，因为这些都是特殊保护的。

Java 提供了 endorsed 技术，用于替换这些类。这个目录下的 jar 包，会比 rt.jar 中的文件，优先级更高，可以被最先加载到。

有哪些地方打破了 Java 的类加载机制?

答：比如tomcat加载兼容版本的WebAppClassLoader、 spi使用上下文加载器，其实是 Application ClassLoader

如何加载一个远程的 .class 文件？怎样加密 .class 文件?

答：无论是远程存储字节码，还是将字节码进行加密，这都是业务需求。要做这些，我们实现一个新的类加载器就可以了。

从栈帧看字节码是如何在 JVM 中进行流转的

java代码到底是如何运行起来的

下面是我们写的代码（ System.out 等模是JRE提供的类库）：

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

javac 进行编译后，会产生 HelloWorld 的字节码， javap 来稍微看一下字节码到底长什么样子：

```
0 getstatic #2 <java/lang/System.out>
3 ldc #3 <Hello World>
5 invokevirtual #4 <java/io/PrintStream.println>
8 return
```

getstatic、ldc、invokevirtual、return 等，就是 opcode（字节码指令）。

继续使用 hexdump 看一下字节码的二进制内容：

```
b2 00 02 12 03 b6 00 04 b1
```

它们的对应关系

```
0xb2   getstatic        获取静态字段的值
0x12   ldc              常量池中的常量值入栈
0xb6   invokevirtual    运行时方法绑定调用方法
0xb1   return           void 函数返回
```

比如 b2 00 02，代表了 getstatic #2 <java/lang/System.out>。

opcode 有一个字节的长度(0~255)，意味着指令集的操作码个数不能操作 256 条。而紧跟在 opcode 后面的是被操作数。比如 b2 00 02，就代表了 getstatic #2 <java/lang/System.out>。

JVM 就是靠解析这些 opcode 和操作数来完成程序的执行的。当我们使用 Java 命令运行 .class 文件的时候，实际上就相当于启动了一个 JVM 进程。

第一阶段总结：

操作数栈：

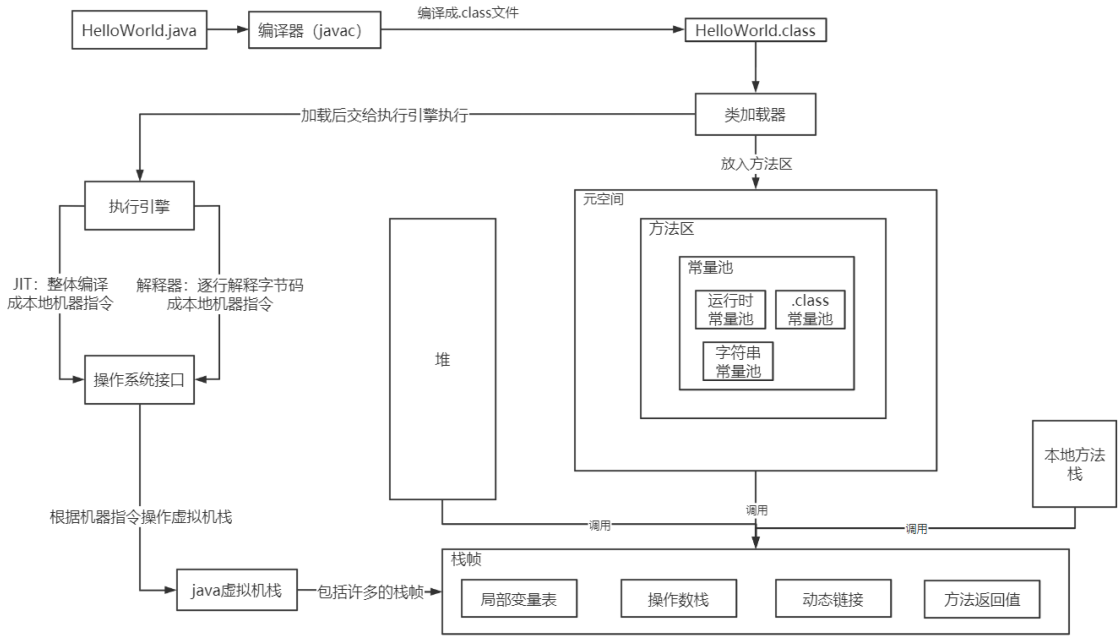
方法在执行的过程中，才会有各种各样的字节码指令往操作数栈中执行入栈和出栈操作。比如在一个方法内部需要执行一个简单的加法运算时，首先需要从操作数栈中将需要执行运算的两个数值出栈，待运算执行完成后，再将运算结果入栈。如下所示：

JVM 会翻译这些字节码，它有两种执行方式。常见的就是解释执行，将 opcode + 操作数翻译成机器代码；另外一种执行方式就是 JIT，也就是我们常说的即时编译，它会在一定条件下将字节码编译成机器码之后再执行。

JIT和解释器的流程区别：

- 解释器：是直接翻译成机器码，抽象的看是这样的：一条一条地读取，解释并且执行字节码指令。
- JIT：先编译成机器码，再执行，抽象的看则是：执行引擎首先按照解释执行的方式来执行，然后在合适的时候，即时编译器把整段字节码**编译成本地代码。**执行本地代码比一条一条进行解释执行的速度快很多。编译后的代码**可以执行的很快，因为本地代码是保存在缓存里的。内置了JIT编译器的JVM都会检查方法的执行频率，如果一个方法的执行频率超过一个特定的值的话，那么这个方法就会被编译成本地代码。**

总结：解释器一次性执行效率更高，JIT多次执行效率更高（因为已经编译好了）



其实整个JVM目的就是在操作虚拟机栈里面的栈帧获取返回值。