链滴

JVM- 类加载机制

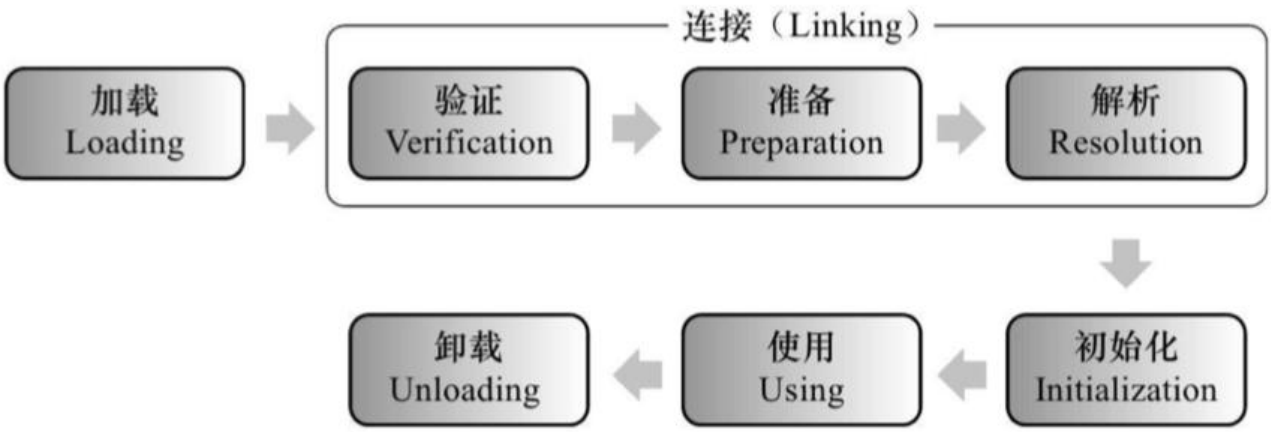
作者：xjlnjut730

原文链接：https://ld246.com/article/1575932482187

来源网站：链滴

今年网上充斥着各种令人焦虑的中年危机新闻，很自然的影响到了我。思来想去，还是需要提升自己核心竞争力。稍微出去面了一下，看看自己竞争力还够不够，发现了问题比我想象中还要严重，随便两个问题就能把我难倒。原因在于随着时间的流逝，好多以前倒背如流的知识点，现在不给点提示，自己压根想不起来。这背后的原因，并不能简单用没有持续学习来搪塞的，因为问到的知识点也没有深入，刚毕业那会儿就很清楚了。这当然是自己工作中疏于总结，疏于温故知新造成的。所以我决定近期开始，对自己的知识体系重新开始梳理。就从这篇JVM的类加载开始吧~~

1 类加载过程



JVM类的生命周期如上图所示。在使用类之前，需要对类进行加载。JVM类加载分为五个部分：加载验证、准备、解析、初始化，其中验证、准备、解析阶段可以统称为连接阶段。

1.1 加载

加载是类加载过程的第一个阶段。**这个阶段会在内存中生成一个代表这个类的java.lang.Class对象，为方法区这个类的各种数据的入口。** Class对象封装了类在方法区内的数据结构，并且向Java程序员供了访问方法区内的数据结构的接口。

注意这里不一定非得要从一个Class文件中获取，这里既可以从zip包中读取（比如jar/war包），孱可在运行时计算生成(动态代理)，也可以由其它文件生成(比如将JSP文件转换成对应的Class类。只需要供给类加载器的字节流符合规范。

1.2 连接

1.2.1 验证

这个阶段的主要目的是确保Class文件的字节流满足当前虚拟机的要求，并且不会危害虚拟机本身的全。验证主要包含四个部分：类文件的结构检查，语义检查，字节码验证，二进制兼容性的验证。

1.2.2 准备

准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量使用的空。这里说的初始值概念，指的是虚拟机规范给定的默认值，而不是代码中申明的初始值（即此时没有始化为真正的初始值）。举个例子：

```
public static int money = 1000;
```

实际上，在准备阶段，给变量money分配的值为0，而非1000。将port赋值为1000是在后续的初始阶段完成的。对应到字节码，就是<clinit>方法执行putstatic指令时完成。如下所示：

```
package cn.xiajl.jvm.classloader;

public class MyTest27 {
    public static int MAX_MONEY = 1000;
}
// 以下为执行javap -verbose cn.xiajl.jvm.classloader.MyTest27的结果，略去无关的细节。
public class cn.xiajl.jvm.classloader.MyTest27
  interfaces: 0, fields: 1, methods: 2, attributes: 1
Constant pool:
   #5 = Utf8               MAX_MONEY
   #6 = Utf8               I
{
  public static int MAX_MONEY;
    descriptor: I
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC

  static {};
    descriptor: ()V
    flags: (0x0008) ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: sipush        1000
         3: putstatic     #2                  // Field MAX_MONEY:I
         6: return
}
```

需要注意的是，如果一个类变量申明为final类型，如下所示：

```
public static final int MAX_MONEY = 1000;
```

此时，准备阶段会直接将MAX_MONEY设为1000，并且将MAX_MONEY生成ConstantValue属性标明为常量。并且在方法中中所有引用到MAX_MONEY的部分，直接替换为常量1000，如下所示：

```
package cn.xiajl.jvm.classloader;

public class MyTest28 {
    public static final int MAX_MONEY = 1000;

    public static void main(String[] args) {
        int currentMoney = MAX_MONEY;
        System.out.println(currentMoney);
    }
}

// 以下为执行javap -verbose cn.xiajl.jvm.classloader.MyTest28的结果，略去无关的细节。
public class cn.xiajl.jvm.classloader.MyTest28
Constant pool:
   ...
   #6 = Utf8               MAX_MONEY
   #7 = Utf8               I
   #8 = Utf8               ConstantValue
   #9 = Integer            1000
```

```
  ...
{
  public static final int MAX_MONEY;
    descriptor: I
    flags: (0x0019) ACC_PUBLIC, ACC_STATIC, ACC_FINAL
    ConstantValue: int 1000

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
         0: sipush        1000
         3: istore_1
         4: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
         7: iload_1
         8: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
        11: return
}
```

1.2.3 解析

解析阶段是指将虚拟机中的符号引用替换为直接引用的过程。符号引用就是class文件中的：CONSTAT_Class_info、CONSTANT_Field_info、Constant_Method_info等类型的常量。

- 符号引用

符号引用与虚拟机实际的布局无关，引用的目标不一定要加载到内存中。各种虚拟机实现的内存布局以各不相同。但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在Java拟机规范的Class文件格式中。

- 直接引用

直接引豕可以是指向目标的指针，相对偏移量或是一个能间接定义到目标的句柄。如果有了直接引用那引用的目标必须已经在内存中存在。

1.3 初始化

初始化阶段是执行类构造器折<clinit>方法的过程。**<clinit>方法是由编译器自动收集类中的类变量赋值操作和静态语句块中的语句合并而成的。** 虚拟机会保证子<clinit>执行之前，父类的<clinit>方已经执行完毕，如果一个类中没有对静态变量赋值也没有静态语句块，那么编译器可以不为这个类生成clinit>方法。

注意，类的初始化只有在以下7种情况下对类的主动使用才会发生：

1. 创建类的实例
2. 访问某个类或接口的静态变量，或者对该静态变量赋值
3. 调用类的静态方法
4. 反射（如Class.forName("com.test.Test")）
5. 初始化一个类的子类
6. Java虚拟机启动时被标明为启动类的类(java com.test.Test)
7. JDK1.7开始提供的动态语言支持：java.lang.invoke.MethodHandle实例的解析结果REF_getStati

，REF_putStatic, REF_invokeStatic句柄对应的类没有初始化，则初始化

尤其注意，以下6种情况不会发生类的初始化：

1. 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。

2. 定义对象数组，不会触发该类的初始化。

3. 常量在编译器间会存入调用类的常量池中，本质上没有直接引用定义常量的类，不会触发定义常量在的类。

4. 通过类名获取Class对象，不会触发类的初始化。

5. 通过Class.forName加载指定类时，如果指定参数initalize为false时，也不会触发类初始化，其实个参数是告诉虚拟机，是否要对灰进行初始化。

6. 通过ClassLoader默认的loadClass方法，也不会触发初始化动作。

1.4 使用

类实例化时，就进行类的使用阶段。此时，JVM会为新对象分配内存，为实例变量赋默认值，为实例量赋正确的初始值。

java编译器为它编译的每一个类都至少生成一个实例化方法。在java的class文件中，这个实例初始化法被称为"<init>"。针对源代码中每一个类的构造方法，java编译器都产生一个<init>方法。

对于这个<init>方法，需要注意的是：对于HotSpot虚拟机，构造块{}中的代码会一并编译进<init>法中，如果有多个<init>方法，构造块中的指令会复制成多份拷贝进每个<init>方法中。

总结一下就是：在编译后，构造块会合并进实例化方法，对象的实例化只需要执行实例化方法即可。

举例如下，我们会看到两个实例化方法编译后的4-13行实际上对应的就是构造块的指令：

```
package cn.xiajl.jvm.classloader;

public class MyTest29 {
    private int money = 10;
    {
        money = 20;
    }

    public MyTest29(int money) {
        this.money = money;
    }

    public MyTest29() {
        this.money = 30;
    }
}

// 以下为执行javap -verbose cn.xiajl.jvm.classloader.MyTest29的结果，略去无关的细节。

public class cn.xiajl.jvm.classloader.MyTest29
  interfaces: 0, fields: 1, methods: 2, attributes: 1
Constant pool:
   #2 = Fieldref          #3.#18         // cn/xiajl/jvm/classloader/MyTest29.money:I
   #5 = Utf8              money
   #6 = Utf8              I
```

```
   #7 = Utf8               <init>
   #8 = Utf8               (I)V
  #14 = Utf8               ()V
  #17 = NameAndType        #7:#14         // "<init>":()V
  #18 = NameAndType        #5:#6          // money:I
{
  public cn.xiajl.jvm.classloader.MyTest29(int);
    descriptor: (I)V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: bipush        10
         7: putfield      #2                  // Field money:I
        10: aload_0
        11: bipush        20
        13: putfield      #2                  // Field money:I
        16: aload_0
        17: iload_1
        18: putfield      #2                  // Field money:I
        21: return

  public cn.xiajl.jvm.classloader.MyTest29();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: bipush        10
         7: putfield      #2                  // Field money:I
        10: aload_0
        11: bipush        20
        13: putfield      #2                  // Field money:I
        16: aload_0
        17: bipush        30
        19: putfield      #2                  // Field money:I
        22: return
}
```

2. 类加载器

虚拟机设计团队把加载动作放到了JVM外部实现，以便让应用程序决定如何获取所需要的类，JVM提了三种类加载器：

1. Bootstrap ClassLoader

即启动类加载器，负责加载JAVA_HOME/lib目录中，或通过-Xbootclasspath参数指定路径中的，被虚拟机认可（按文件名识别，如rt.jar)的类。

2. Extension ClassLoader

即扩展类加载器，负责加载JAVA_HOME/lib/ext目录中的，或通过java.ext.dirs变量指定路径中的类。

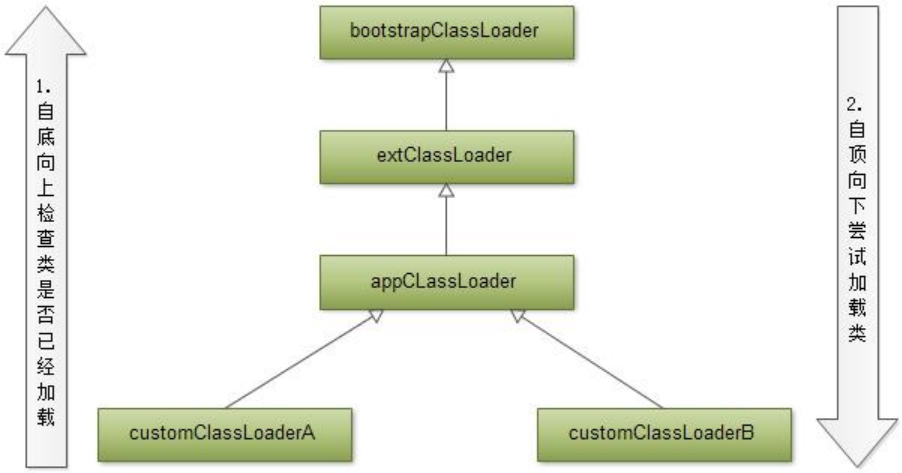
3. Application ClassLoader

即应用类加载器，负责加载用户路径(classpath)上的类库。

除了以上三种JVM自带的类加载器之后，程序中也可通过继承java.lang.ClassLoader来实现自己定义类加载器。

2.1 双亲委派

JVM通过双亲委派模型来进行类的加载：



当一个类加载器收到类加载请求，他首先不会尝试自己去加载这个类，而把这个请求委派给父类去完，每个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载器加载。只有当父类加器反馈自己无法完成这个请求时（在它的加载路径下没有找到所需加载的Class)，子类加载器才会尝自己去加载。

父类委托机制的优点是能够提供软件系统的安全性。因为在此机制下，用户自定义的类加载器不可能载应由其父加载器加载的可靠类，从而防止不可靠甚至恶意的代码代替由父加载器加载的可靠代码。如，java.lang.Object类总是由根类加载器加载，其它自定义的类加载器都不可能加载含有恶意代码的ava.lang.Object类。

2.2 命名空间

每个类加载器都有自己的命令空间，命令空间由该加载器及所有父加载器所加载的类组成。

在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类。在不同的命名空间中有可能会出现类的完整名字（包括类的包名）相同的两个类。

同一个命名空间内的类是相互可见的。子加载器的命名空间包含所有父加载器的命名空间，因此由子载器加载的类能够看见父加载器加载的类。例如应用类加载器加载的类能看见启动类加载器加载的类。

由父加载器加载的类不能看见子加载器加载的类。

如果两个加载器之间没有直接或间接的父子关系，那么它们相互不可见。

2.3 线程上下文加载器

在双亲委托模型下，类加载是由下至上的，即下层的类加载器会委托上进行加载。但是对于SPI来说有些接口是由Java核心库所提供的， 而Java核心库是由启动类加载器，而这些接口的实现却来自于同的jar包（厂商提供），Java的启动类加载器是不会加载其他来源的jar包， 这样传统的双亲委托模就无法满足SPI的要求，而通过给当前线程设置上下文类加载器，就可以由设置的线程上下文类加载来实现对于接口实现类的加载。

线程上下文类加载器是从JDK1.2开始引入的，类Thread中的getContextClassLoader()与setContextClassLoader(ClassLoader cl) 分别用来获取和设置上下文类加载器。

如果没有通过setContextClassLoader（ClassLoader cl)进行设置的话，线程将继承其父线程的上下类加载器，Java应用运行时的初始线程的上下文类加载器是应用类加载器。在线程中运行的代码可以过该类加载器来加载类与资源。

父Classloader可以使用当前线程Thread.currentThread().getContextClassLoader()所指定的classloader加载的类。这就改变了父ClassLoader不使用子ClassLoader或是其他没有直接父子关系的ClassLoader加载的类的情况，即改变了双亲委托模型。

线程上下文类加载器的一般使用模式如下所示，即：获取->使用->还原：

```
// 获取 - 使用 - 还原
ClassLoader classLoader = Thread.currentThread().getContxtClassLoader();
try {
    Thread.currentThread().setContextClassLoader(targetTccl);
    myMethod();
} finally {
    Thread.currentThread().setContextClassLoader(classLoader);
}
```

myThread里面则调用了Thread.currentThread().getContextClassLoader(), 获取当前线程的上下文加载器做某些事情。

如果一个类由类加载器A加载，那么这个类的依赖类也是由相同的类加载器加载的（如果该依赖类之没有被加载过的话）

ContextClassLoader的作用就是为了破坏Java的类加载委托机制。

当高层提供了统一的接口让低层实现，同时又要在高层加载(或实例化）低层的类时，就必须通过线程下文类加载器来帮助高层的ClassLoader提取并加载该类。