二

03 Java虚拟机是如何加载Java类的

听我的意大利同事说，他们那边有个习俗，就是父亲要帮儿子盖栋房子。

这事要放在以前还挺简单，亲朋好友搭把手，盖个小砖房就可以住人了。现在呢，整个过程要耗费好久的时间。首先你要请建筑师出个方案，然后去市政部门报备、验证，通过后才可以开始盖房子。盖好房子还要装修，之后才能住人。

盖房子这个事，和 Java 虚拟机中的类加载还是挺像的。从 class 文件到内存中的类，按先后顺序需要经过加载、链接以及初始化三大步骤。其中，链接过程中同样需要验证；而内存中的类没有经过初始化，同样不能使用。那么，是否所有的 Java 类都需要经过这几步呢？

我们知道 Java 语言的类型可以分为两大类：基本类型（primitive types）和引用类型（reference types）。在上一篇中，我已经详细介绍过了 Java 的基本类型，它们是由 Java 虚拟机预先定义好的。

至于另一大类引用类型，Java 将其细分为四种：类、接口、数组类和泛型参数。由于泛型参数会在编译过程中被擦除（我会在专栏的第二部分详细介绍），因此 Java 虚拟机实际上只有前三种。在类、接口和数组类中，数组类是由 Java 虚拟机直接生成的，其他两种则有对应的字节流。

说到字节流，最常见的形式要属由 Java 编译器生成的 class 文件。除此之外，我们也可以在程序内部直接生成，或者从网络中获取（例如网页中内嵌的小程序 Java applet）字节流。这些不同形式的字节流，都会被加载到 Java 虚拟机中，成为类或接口。为了叙述方便，下面我就用“类”来统称它们。

无论是直接生成的数组类，还是加载的类，Java 虚拟机都需要对其进行链接和初始化。接下来，我会详细给你介绍一下每个步骤具体都在干些什么。

加载

加载，是指查找字节流，并且据此创建类的过程。前面提到，对于数组类来说，它并没有对应的字节流，而是由 Java 虚拟机直接生成的。对于其他的类来说，Java 虚拟机则需要借助类加载器来完成查找字节流的过程。

以盖房子为例，村里的 Tony 要盖个房子，那么按照流程他得先找个建筑师，跟他说想要设计一个房型，比如说“一房、一厅、四卫”。你或许已经听出来了，这里的房型相当于类，而建筑师，就相当于类加载器。

村里有许多建筑师，他们等级森严，但有着共同的祖师爷，叫启动类加载器（bootstrap class loader）。启动类加载器是由 C++ 实现的，没有对应的 Java 对象，因此在 Java 中只能用 null 来指代。换句话说，祖师爷不喜欢像 Tony 这样的小角色来打扰他，所以谁也没有祖师爷的联系方式。

除了启动类加载器之外，其他的类加载器都是 java.lang.ClassLoader 的子类，因此有对应的 Java 对象。这些类加载器需要先由另一个类加载器，比如说启动类加载器，加载至 Java 虚拟机中，方能执行类加载。

村里的建筑师有一个潜规则，就是接到单子自己不能着手干，得先给师傅过过目。师傅不接手的情况下，才能自己来。在 Java 虚拟机中，这个潜规则有个特别的名字，叫双亲委派模型。每当一个类加载器接收到加载请求时，它会先将请求转发给父类加载器。在父类加载器没有找到所请求的类的情况下，该类加载器才会尝试去加载。

在 Java 9 之前，启动类加载器负责加载最为基础、最为重要的类，比如存放在 JRE 的 lib 目录下 jar 包中的类（以及由虚拟机参数 -Xbootclasspath 指定的类）。除了启动类加载器之外，另外两个重要的类加载器是扩展类加载器（extension class loader）和应用类加载器（application class loader），均由 Java 核心类库提供。

扩展类加载器的父类加载器是启动类加载器。它负责加载相对次要、但又通用的类，比如存放在 JRE 的 lib/ext 目录下 jar 包中的类（以及由系统变量 java.ext.dirs 指定的类）。

应用类加载器的父类加载器则是扩展类加载器。它负责加载应用程序路径下的类。（这里的应用程序路径，便是指虚拟机参数 -cp/-classpath、系统变量 java.class.path 或环境变量 CLASSPATH 所指定的路径。）默认情况下，应用程序中包含的类便是由应用类加载器加载的。

Java 9 引入了模块系统，并且略微更改了上述的类加载器1。扩展类加载器被改名为平台类加载器（platform class loader）。Java SE 中除了少数几个关键模块，比如说 java.base 是由启动类加载器加载之外，其他的模块均由平台类加载器所加载。

除了由 Java 核心类库提供的类加载器外，我们还可以加入自定义的类加载器，来实现特殊的加载方式。举例来说，我们可以对 class 文件进行加密，加载时再利用自定义的类加载器对其解密。

除了加载功能之外，类加载器还提供了命名空间的作用。这个很好理解，打个比方，咱们这个村不讲究版权，如果你剽窃了另一个建筑师的设计作品，那么只要你标上自己的名字，这

两个房型就是不同的。

在 Java 虚拟机中，类的唯一性是由类加载器实例以及类的全名一同确定的。即便是同一串字节流，经由不同的类加载器加载，也会得到两个不同的类。在大型应用中，我们往往借助这一特性，来运行同一个类的不同版本。

链接

链接，是指将创建成的类合并至 Java 虚拟机中，使之能够执行的过程。它可分为验证、准备以及解析三个阶段。

验证阶段的目的，在于确保被加载类能够满足 Java 虚拟机的约束条件。这就好比 Tony 需要将设计好的房型提交给市政部门审核。只有当审核通过，才能继续下面的建造工作。

通常而言，Java 编译器生成的类文件必然满足 Java 虚拟机的约束条件。因此，这部分我留到讲解字节码注入时再详细介绍。

准备阶段的目的，则是为被加载类的静态字段分配内存。Java 代码中对静态字段的具体初始化，则会在稍后的初始化阶段中进行。过了这个阶段，咱们算是盖好了毛坯房。虽然结构已经完整，但是在没有装修之前是不能住人的。

除了分配内存外，部分 Java 虚拟机还会在此阶段构造其他跟类层次相关的数据结构，比如说用来实现虚方法的动态绑定的方法表。

在 class 文件被加载至 Java 虚拟机之前，这个类无法知道其他类及其方法、字段所对应的具体地址，甚至不知道自己方法、字段的地址。因此，每当需要引用这些成员时，Java 编译器会生成一个符号引用。在运行阶段，这个符号引用一般都能够无歧义地定位到具体目标上。

举例来说，对于一个方法调用，编译器会生成一个包含目标方法所在类的名字、目标方法的名字、接收参数类型以及返回值类型的符号引用，来指代所要调用的方法。

解析阶段的目的，正是将这些符号引用解析成为实际引用。如果符号引用指向一个未被加载的类，或者未被加载类的字段或方法，那么解析将触发这个类的加载（但未必触发这个类的链接以及初始化。）

如果将这段话放在盖房子的语境下，那么符号引用就好比“Tony 的房子”这种说法，不管它存在不存在，我们都可以用这种说法来指代 Tony 的房子。实际引用则好比实际的通讯地址，如果我们想要与 Tony 通信，则需要启动盖房子的过程。

Java 虚拟机规范并没有要求在链接过程中完成解析。它仅规定了：如果某些字节码使用了符号引用，那么在执行这些字节码之前，需要完成对这些符号引用的解析。

初始化

在 Java 代码中，如果要初始化一个静态字段，我们可以在声明时直接赋值，也可以在静态代码块中对其赋值。

如果直接赋值的静态字段被 final 所修饰，并且它的类型是基本类型或字符串时，那么该字段便会被 Java 编译器标记成常量值（ConstantValue），其初始化直接由 Java 虚拟机完成。除此之外的直接赋值操作，以及所有静态代码块中的代码，则会被 Java 编译器置于同一方法中，并把它命名为 < clinit >。

类加载的最后一步是初始化，便是为标记为常量值的字段赋值，以及执行 < clinit > 方法的过程。Java 虚拟机会通过加锁来确保类的 < clinit > 方法仅被执行一次。

只有当初始化完成之后，类才正式成为可执行的状态。这放在我们盖房子的例子中就是，只有当房子装修过后，Tony 才能真正地住进去。

那么，类的初始化何时会被触发呢？JVM 规范枚举了下述多种触发情况：

1. 当虚拟机启动时，初始化用户指定的主类；
2. 当遇到用以新建目标类实例的 new 指令时，初始化 new 指令的目标类；
3. 当遇到调用静态方法的指令时，初始化该静态方法所在的类；
4. 当遇到访问静态字段的指令时，初始化该静态字段所在的类；
5. 子类的初始化会触发父类的初始化；
6. 如果一个接口定义了 default 方法，那么直接实现或者间接实现该接口的类的初始化，会触发该接口的初始化；
7. 使用反射 API 对某个类进行反射调用时，初始化这个类；
8. 当初次调用 MethodHandle 实例时，初始化该 MethodHandle 指向的方法所在的类。

```
public class Singleton {
  private Singleton() {}
  private static class LazyHolder {
    static final Singleton INSTANCE = new Singleton();
  }
  public static Singleton getInstance() {
    return LazyHolder.INSTANCE;
  }
}
```

我在文章中贴了一段代码，这段代码是在著名的单例延迟初始化例子中2，只有当调用 Singleton.getInstance 时，程序才会访问 LazyHolder.INSTANCE，才会触发对 LazyHolder 的初始化（对应第 4 种情况），继而新建一个 Singleton 的实例。

由于类初始化是线程安全的，并且仅被执行一次，因此程序可以确保多线程环境下有且仅有一个 Singleton 实例。

总结与实践

今天我介绍了 Java 虚拟机将字节流转化为 Java 类的过程。这个过程可分为加载、链接以及初始化三大步骤。

加载是指查找字节流，并且据此创建类的过程。加载需要借助类加载器，在 Java 虚拟机中，类加载器使用了双亲委派模型，即接收到加载请求时，会先将请求转发给父类加载器。

链接，是指将创建成的类合并至 Java 虚拟机中，使之能够执行的过程。链接还分验证、准备和解析三个阶段。其中，解析阶段为非必须的。

初始化，则是为标记为常量值的字段赋值，以及执行 < clinit > 方法的过程。类的初始化仅会被执行一次，这个特性被用来实现单例的延迟初始化。

今天的实践环节，你可以来验证一下本篇中的理论知识。

通过 JVM 参数 -verbose:class 来打印类加载的先后顺序，并且在 LazyHolder 的初始化方法中打印特定字样。在命令行中运行下述指令（不包含提示符 $）：

```
$ echo '
public class Singleton {
  private Singleton() {}
  private static class LazyHolder {
    static final Singleton INSTANCE = new Singleton();
    static {
      System.out.println("LazyHolder.<clinit>");
```

```
    }
  }
  public static Object getInstance(boolean flag) {
    if (flag) return new LazyHolder[2];
    return LazyHolder.INSTANCE;
  }
  public static void main(String[] args) {
    getInstance(true);
    System.out.println("----");
    getInstance(false);
  }
}' > Singleton.java
$ javac Singleton.java
$ java -verbose:class Singleton
```

问题 1：新建数组（第 11 行）会导致 LazyHolder 的加载吗？会导致它的初始化吗？

在命令行中运行下述指令（不包含提示符 $）：

```
$ java -cp /path/to/asmtools.jar org.openjdk.asmtools.jdis.Main Singleton\$LazyHold
$ awk 'NR==1,/stack 1/{sub(/stack 1/, "stack 0")} 1' Singleton\$LazyHolder.jasm.1 >
$ java -cp /path/to/asmtools.jar org.openjdk.asmtools.jasm.Main Singleton\$LazyHold
$ java -verbose:class Singleton
```

上一页 下一页