第五章

加载、链接和初始化

Java 虚拟机动态加载、链接和初始化类和接口。加载是查找具有特定名称的类或接口类型的二进制表示，并根据该二进制表示创建类或接口的过程。链接是获取一个类或接口，并将其合并到 Java 虚拟机的运行时状态中，以便执行它的过程。类或接口的初始化包括执行类或接口的初始化方法<clinit>(§2.9.2)。

在本章中，§5.1 描述了 Java 虚拟机如何从类或接口的二进制表示中获取符号引用。§5.2 解释了加载、链接和初始化过程是如何首先由 Java 虚拟机发起的。§5.3 规定了类加载器如何加载类和接口的二进制表示形式，以及如何创建类和接口。链接在§5.4 中有描述。§5.5 详细介绍了如何初始化类和接口。§5.6 引入了绑定本地方法的概念。最后，§5.7 描述了 Java 虚拟机何时退出。

5.1 运行时常量池

Java 虚拟机为每个类和接口维护一个运行时常量池(§2.5.5)。这种数据结构满足了传统编程语言实现的符号表的许多目的。类或接口的二进制表示中的 constant_pool(§4.4)表用于在类或接口创建时构造运行时常量池(§5.3)。运行时常量池中有两种类型的条目:符号引用(稍后可以解析)和静态常量(无需进一步处理)。

运行时常量池中的符号引用根据每个条目的结构派生自 constant_pool 表中的条目:

- 对类或接口的符号引用派生自 CONSTANT_Class_info 结构体(§4.4.1)。这样的引用以下列形式给出类或接口的名称:
  - 对于非数组类或接口，名称是类或接口的二进制名称(§4.2.1)。
  - 对于 n 维的数组类，名称以 n 次 ASCII[字符开头，后面是元素类型的表示:
    > 如果元素类型是原生类型，则由相应的字段描述符表示(§4.3.2)。
    > 否则，如果元素类型是引用类型，则它由 ASCII L 字符后接元素类型的二进制名称，再接 ASCII;字符。

  无论本章何时引用类或接口的名称，都应该理解为上面的形式。(这也是 Class.getName 方法返回的形式。)
- 对类或接口字段的符号引用派生自 CONSTANT_Fieldref_info 结构体(§4.4.2)。这种引用提供了字段的名称和描述符，以及对要在其中找到字段的类或接口的符号引用。
- 类方法的符号引用源自 CONSTANT_Methodref_info 结构（§4.4.2）。这样的引用给出了方法的名称和描述符，以及对要在其中找到方法的类的符号引用。

- 接口的方法的符号引用派生自 CONSTANT_InterfaceMethodref_info 结构体(§4.4.2)。这样的引用提供了接口方法的名称和描述符，以及对要在其中找到方法的接口的符号引用。
- 方法句柄的符号引用派生自CONSTANT_MethodHandle_info结构体(§4.4.8)。这样的引用提供对类或接口的字段、类的方法或接口的方法的符号引用，这取决于方法句柄的类型。
- 方法类型的符号引用派生自CONSTANT_MethodType_info结构体(§4.4.9)。这样的引用给出了一个方法描述符(§4.3.3)。
- 对动态计算的常数的符号引用派生自 CONSTANT_Dynamic_info 结构体(§4.4.10)。这样的引用给出:
  - 对方法句柄的符号引用，将调用该句柄来计算常量的值;
  - 符号引用和静态常量的序列，在调用方法句柄时作为静态参数;
  - 非限定名称和字段描述符。
- 对动态计算的调用站点的符号引用派生自 CONSTANT_InvokeDynamic_info 结构(§4.4.10)。这样的引用给出:
  - 一个方法句柄的符号引用，它将在 invokedynamic 指令(§invokedynamic)的过程中被调用，以计算 java.lang.invoke.CallSite 的实例;
  - 符号引用和静态常量的序列，在调用方法句柄时作为静态参数;
  - 非限定名称和方法描述符。

运行时常量池中的静态常量也根据每个条目的结构从 constant_pool 表中的条目派生:

- 字符串常量是对类 String 实例的引用，并从 CONSTANT_String_info 结构派生（§4.4.3）。为了派生字符串常量，Java 虚拟机检查由 CONSTANT_String_info 结构给出的代码点序列:
  - 如果 String.intern 方法之前已经在一个 String 类的实例上调用过，该实例包含的 Unicode 代码点序列与 CONSTANT_String_info 结构给出的相同，那么该字符串常量就是对同一个 String 类实例的引用。
  - 否则，将创建一个新的字符串类实例，其中包含由CONSTANT_String_info结构给出的 Unicode 码位序列。字符串常量是对新实例的引用。最后，在新实例上调用 String.intern 方法。
- 数值常量派生自 CONSTANT_Integer_info, CONSTANT_Float_info, CONSTANT_Long_info, 和 CONSTANT_Double_info 结构(§4.4.4, §4.4.5)。

  注意，CONSTANT_Float_info 结构表示 IEEE 754 单精度格式的值，CONSTANT_Double_info 结构表示 IEEE 754 双精度格式的值。因此，从这些结构派生的数值常量必须是可以分别使用 IEEE 754 单精度格式和双精度格式表示的值。

constant_pool 表中其余的结构-描述性的结构 CONSTANT_NameAndType_info,

CONSTANT_Module_info, 和 CONSTANT_Package_info, 以及基本结构 CONSTANT_Utf8_info-仅在构造运行时常量池时间接使用。运行时常量池中没有直接对应于这些结构的条目。

运行时常量池中的一些条目是可加载的，这意味着:

- 它们可能被 ldc 指令族推入栈(§ldc, §ldc_w, §ldc2_w)。
- 它们可以是 bootstrap 方法的静态参数，用于动态计算常量和调用站点(§5.4.3.6)。

运行时常量池中的一个条目是可加载的，如果它派生自 constant_pool 表中可加载的条目(参见表 4.4-C)。相应地，运行时常量池中的以下条目是可加载的:

- 对类和接口的符号引用
- 对方法句柄的符号引用
- 对方法类型的符号引用
- 对动态计算常量的符号引用
- 静态常量

5.2 Java 虚拟机启动

Java 虚拟机通过使用引导类加载器(§5.3.1)或用户定义的类加载器(§5.3.2)创建一个初始类或接口来启动。然后，Java 虚拟机链接初始类或接口，对其进行初始化，并调用公共静态方法 void main(String[])。此方法的调用将推动所有进一步的执行。执行构成 main 方法的 Java 虚拟机指令可能会导致链接(并创建)其他类和接口，以及调用其他方法。

初始类或接口以依赖于实现的方式指定。例如，初始类或接口可以作为命令行参数提供。或者，Java 虚拟机的实现本身可以提供一个初始类，用于设置加载应用程序的类加载器。初始类或接口的其他选择是可能的，只要它们与前一段给出的规范一致。

5.3 创建和加载

创建一个名为 N 的类或接口，需要在 Java 虚拟机的方法区(§2.5.4)构造一个特定于实现的 C 内部表示。

类或接口的创建是由另一个类或接口 D 触发的，其运行时常量池通过名称 N 象征性地引用 C(§5.4.3.1)。如果 N 不表示数组类，那么 Java 虚拟机将依赖于类加载器来定位一个名为 N 的类或接口的二进制表示(§4.1)。一旦类加载器找到了二进制表示，它就转而依赖 Java 虚拟机从二进制表示派生类或接口 C，然后在方法区中创建 C。数组类没有外部二进制表示;它们是由 Java 虚拟机通过不同的过程创建的。

类或接口的创建也可以通过 D 调用某些 Java SE 平台类库(§2.12)中的方法(如反射)来触发。

有两种类加载器:Java 虚拟机提供的引导类加载器和用户定义的类加载器。每个用户定义的类加载器都是抽象类 ClassLoader 的一个子类的实例。应用程序使用用户定义的类加载器来扩展 Java 虚拟机动态创建类的方式。用户定义的类加载器可用于创建源自用户定义源的类。例如，可以通过网络下载类、动态生成类或从加密文件中提取类。

当 Java 虚拟机请求类加载器 L 定位名为 N 的类或接口的二进制表示时，L 加载用 N 表示的类或接口 C。L 可以直接加载 C，通过定位一个二进制表示，并要求 Java 虚拟机从二进制表示派生和创建 C。或者，L 也可以通过委托给另一个直接或间接加载 C 的类加载器来间接加载 C。

如果 L 直接加载 C，我们说 L 定义了 C，或者，L 是 C 语言的定义加载器。

无论 L 直接或间接加载 C，我们都说 L 启动了 C 的加载，或者，等价地说，L 是 C 的初始加载器。

由于类加载器委托，根据 Java 虚拟机的请求启动加载的加载器 L1 可能与通过定义类或接口完成加载的加载器 L2 不同。在这种情况下，我们说每个 L1 和 L2 都启动了 C 的加载，或者，等效地说，每个 L1 和 L2 都是 C 的初始加载器。L1 和 L2 之间的委托链中的任何加载器都不被认为是 C 的启动加载器。

我们有时会使用下面的符号来表示一个类或接口，而不是使用像 C 或 D 这样的标识符:

- <N, L_d> - 其中 N 表示类或接口的名称，L_d表示类或接口的定义加载器。
- N^{Li} - 其中 N 表示类或接口的名称，L_i表示类或接口的启动加载器。

  很明显，加载类或接口是 Java 虚拟机和类加载器(或者多个类加载器，如果发生委托的话)之间的共同努力。加载的最终结果是 Java 虚拟机在其方法区中创建一个类或接口，因此通常可以方便地说，类或接口被加载并由此创建。

  加载的复杂来回性质，加上用户定义的类加载器显示任意行为的能力，意味着可以在 Java 虚拟机创建类或接口之后，但在参与加载的每个类加载器完成之前抛出异常。该规范说明了在加载和创建类或接口的过程中经常出现的异常。

Java 虚拟机使用以下三个过程之一来创建类或接口 C，在类或接口 D 的运行时常量池中用名称 N 表示:

• 如果 N 表示非数组类或接口，D 由引导类加载器定义，那么引导类加载器将启动对 C 的加载(§5.3.1)。

• 如果 N 表示非数组类或接口，而 D 是由用户定义的类加载器定义的，那么该用户定义的类加载器将启动对 C 的加载(§5.3.2)。

• 如果 N 表示一个数组类，那么 Java 虚拟机创建一个数组类 C，用 N 表示，与 D 的定义加载器(§5.3.3)相关联。

虽然 D 的定义加载器在创建数组类的过程中是相关的，但它不用于装入并因此创建数组类。

如果在加载类或接口期间发生错误——无论是在类加载器定位二进制表示时，还是在 Java

虚拟机从它派生和创建类时——则必须在程序中(直接或间接)使用正在加载的类或接口的某个点抛出错误。

一个表现良好的类加载器应该维护三个属性:

- 给定相同的名称，一个好的类加载器应该总是返回相同的 Class 对象。
- 如果一个类加载器 L1 将类 C 的加载委托给另一个加载器 L2，那么对于作为 C 的直接超类或直接超接口，或作为 C 中的字段的类型，或作为 C 中的方法或构造函数的形式参数的类型，或作为 C 中的方法的返回类型的任何类型 T, L1 和 L2 应该返回相同的 Class 对象。
- 如果用户定义的类加载器预取类和接口的二进制表示形式，或者将一组相关的类加载在一起，那么它必须只在程序中不进行预取或组加载就可能出现加载错误的地方反映加载错误。

创建后，类或接口不是由它的名字单独决定的，而是由一对决定的:它的二进制名称(§4.2.1)和它的定义加载器。每个这样的类或接口都属于单个运行时包。类或接口的运行时包由包名和类或接口的定义加载器决定。

5.3.1 使用引导类加载器加载

使用引导类加载器加载和创建 N 表示的非数组类或接口 C 的过程如下。

首先，Java 虚拟机确定引导类加载器是否已经被记录为 N 表示的类或接口的启动加载器。如果是这样的话，这个类或接口是 C，不需要加载或创建类。

否则，Java 虚拟机将参数 N 传递给引导类加载器上的方法调用。为了加载 C，引导类加载器以一种平台相关的方式定位 C 的声明表示，然后要求 Java 虚拟机使用引导类加载器从声明表示中派生出一个由 N 表示的类或接口 C，然后通过§5.3.5 的算法创建 C。

通常，类或接口将使用分层文件系统中的文件来表示，类或接口的名称将被编码到文件的路径名中，以帮助定位它。

如果没有找到 C 的声明表示，引导类加载器就会抛出 ClassNotFoundException 异常。然后，加载和创建 C 的过程会失败，产生一个 NoClassDefFoundError，其原因是 ClassNotFoundException。

如果找到了 C 的声明表示，但是从声明表示中导出 C 失败了，那么加载和创建 C 的过程也会因为同样的原因失败。

否则，加载并创建 C 的过程将成功。

5.3.2 使用用户定义的类加载器进行加载

使用用户定义的类加载器 L 加载和创建由 N 表示的非数组类或接口 C 的过程如下。

首先，Java 虚拟机确定 L 是否已经被记录为由 N 表示的类或接口的启动加载器。如果是这样，这个类或接口就是 C，不需要加载或创建类。

否则，Java 虚拟机调用 L 上 ClassLoader 类的 loadClass 方法，传递类或接口的名称 N。L 必须执行以下两个操作之一来加载并创建类或接口 C:

1. 类加载器 L 可以直接加载 C。这是通过获取一个字节数组来表示 C 为 ClassFile 结构(§4.1)，然后调用 ClassLoader 类的 defineClass 方法来实现的。调用 defineClass 会导致 Java 虚拟机使用 L 从字节数组中派生一个由 N 表示的类或接口 C，然后通过§5.3.5 的算法创建 C。L 应该使用 defineClass 的结果作为 loadClass 的结果。

2. 类加载器 L 可以通过将 C 的加载委托给其他类加载器 L'来间接装入 C。这是通过将参数 N 传递给 L'上的方法调用(通常是 ClassLoader 类的 loadClass 方法)来完成的。L 应该使用该方法的结果作为 loadClass 的结果。

以下规则适用于任何操作:

• 如果类加载器不能找到 N 表示的类或接口的声明表示，它必须抛出 ClassNotFoundException。然后，加载和创建 C 的过程会失败，产生一个 NoClassDefFoundError，其原因是 ClassNotFoundException。

• 如果类加载器找到了 C 的声明表示，但是从声明表示派生 C 失败了，那么加载和创建 C 的过程也会因为同样的原因失败。

• 如果类加载器抛出 ClassNotFoundException 以外的异常，那么加载和创建 C 的过程也会因为同样的原因失败。

如果 L 上的 loadClass 调用有结果，那么:

• 如果结果为 null，或者结果是名称不是 N 的类或接口，则该结果将被丢弃，加载和创建过程将失败，并产生 NoClassDefFoundError。

• 否则，结果就是创建的类或接口 C。Java 虚拟机记录 L 是 C 的启动加载器(§5.3.4)。加载和创建 C 的过程成功。

从 JDK 1.1 开始，Oracle 的 Java 虚拟机实现在类加载器上调用一个参数的 loadClass 方法来加载类或接口。loadClass 的参数是要加载的类或接口的名称。loadClass 方法还有一个有两个参数的版本，其中第二个参数是一个布尔值，表示是否要链接类或接口。在 JDK 1.0.2 中只提供了两个参数的版本，Oracle 的 Java 虚拟机实现依赖于它来链接加载的类或接口。从 JDK 1.1 开始，Oracle 的 Java 虚拟机实现直接链接类或接口，而不依赖于类加载器。

5.3.3 创建数组类

下面的步骤用于创建与类加载器 L 关联的名称 N 表示的数组类 C。L 可以是引导类装入器或用户定义的类装入器。L 可以是引导程序类加载器，也可以是用户定义的类加载器。

首先，Java 虚拟机确定 L 是否已经被记录为与 N 具有相同组件类型的数组类的启动加载器。如果是这样，这个类就是 C，不需要创建数组类。

否则，按如下步骤创建 C:

1. 如果组件类型是 reference 类型，则本节(§5.3)的算法会递归地使用 L 来加载并创建组件类型 C。

2. Java 虚拟机使用指定的组件类型和维数创建一个新的数组类。

如果组件类型是 reference 类型，Java 虚拟机将 C 标记为将组件类型的定义加载器作为其定义加载器。否则，Java 虚拟机将 C 标记为将引导类加载器作为其定义加载器。

在任何情况下，Java 虚拟机然后记录 L 是 C 的启动加载器(§5.3.4)。

如果组件类型是 reference 类型，则数组类的可访问性由其组件类型的可访问性决定(§5.4.4)。否则，所有类和接口都可以访问数组类。

5.3.4 加载约束

在存在类加载器的情况下确保类型安全连接需要特别小心。当两个不同的类加载器开始加载由 N 表示的类或接口时，名称 N 可能表示每个加载器中的不同类或接口。

当一个类或接口 C = <N_1, L_1>对另一个类或接口 D = <N_2, L_2>的字段或方法进行符号引用时，符号引用包括一个描述符，指定字段的类型，或方法的返回类型和参数类型。当 L_1 加载和 L_2 加载时，字段或方法描述符中提到的任何类型名称 N 都表示相同的类或接口，这是至关重要的。

为了确保这一点，Java 虚拟机在准备(§5.4.2)和解析(§5.4.3)期间强加了形式为 $N^{L1} = N^{L2}$ 的加载约束。为了实施这些约束，Java 虚拟机将在特定的时间(参见§5.3.1，§5.3.2，§5.3.3 和§5.3.5)记录特定的加载器是特定类的启动加载器。在记录加载器是类的启动加载器之后，Java 虚拟机必须立即检查是否违反了任何加载约束。如果是这样，记录被收回，Java 虚拟机抛出一个 LinkageError，导致记录发生的加载操作失败。

同样，在施加了加载约束(见§5.4.2、§5.4.3.2、§5.4.3.3 和§5.4.3.4)后，Java 虚拟机必须立即检查是否有任何加载约束被违反。如果是这样，则收回新施加的加载约束，Java 虚拟机抛出一个 LinkageError，导致施加约束的操作(解析或准备，视情况而定)失败。

这里描述的情况只是 Java 虚拟机检查是否违反了任何加载约束的情况。当且仅当以下四个条件都满足时，就违反了加载约束:

- 存在一个加载器 L，这样 L 就被 Java 虚拟机记录为一个名为 N 的类 C 的启动加载器。
- 存在一个加载器 L'，这样 L'就被 Java 虚拟机记录为名为 N 的类 C '的启动加载器。
- 由施加的约束集合(传递闭包)定义的等价关系意味着 $N^{L} = N^{L'}$。
- C≠C '。

关于类加载器和类型安全的详细讨论超出了本规范的范围。关于更全面的讨论，读者可以参考 Sheng Liang 和 Gilad Bracha 的《Java 虚拟机中的动态类加载》(1998 年 ACM 面向对象编程系统、语言和应用 SIGPLAN 会议会议记录)。

5.3.5 从 class 文件表示派生类

以下步骤使用类加载器 L 从声明的 class 文件格式表示派生出由 N 表示的非数组类或接口 C。

1. 首先，Java 虚拟机确定 L 是否已经被记录为 N 所表示的类或接口的启动加载器。如果

是，这个派生尝试是无效的，并且抛出一个 LinkageError。

2. 否则，Java 虚拟机将尝试解析声明的表示。所声称的表示实际上可能不是 C 的有效表示，所以派生必须检测以下问题:

- 如果声明的表示不是 ClassFile 结构(§4.1，§4.8)，派生会抛出 ClassFormatError。
- 否则，如果声明的表示不是受支持的主要或次要版本(§4.1)，派生会抛出 UnsupportedClassVersionError。

  JDK 1.2 中引入了 UnsupportedClassVersionError，它是 ClassFormatError 的一个子类，这样当试图加载一个使用不支持的 class 文件格式表示的类时，可以很容易地识别出 ClassFormatError。在 JDK 1.1 及更早的版本中，当不支持的版本出现时，会抛出 NoClassDefFoundError 或 ClassFormatError 的实例，这取决于类是由系统类加载器加载的还是用户定义的类加载器加载的。

- 否则，如果声明的表示实际上并不表示名为 N 的类或接口，派生将抛出 NoClassDefFoundError。

  当声明的表示有一个 this_class 项指定了一个不是 N 的名称，或者一个 access_flags 项设置了 ACC_MODULE 标志时，就会发生这种情况。

3. 如果 C 有一个直接超类，C 对其直接超类的符号引用将使用§5.4.3.1 的算法进行解析。注意，如果 C 是一个接口，它必须有 Object 作为它的直接超类，这个超类必须已经被加载。只有 Object 没有直接超类。

因类或接口解析失败而引发的任何异常都可以因派生而引发。此外，派生必须检测以下问题:

- 如果 C 的任何超类是 C 本身，派生就会抛出 ClassCircularityError。
- 否则，如果被命名为 C 的直接超类的类或接口实际上是一个接口或 final 类，派生将抛出 IncompatibleClassChangeError。
- 否则，如果被命名为 C 的直接超类的类有一个 PermittedSubclasses 属性(§4.7.31)，并且下列任何一项为 true，派生就会抛出一个 IncompatibleClassChangeError:
  - 超类与 C 在不同的运行时模块中(§5.3.6)。
  - C 没有设置 acc_public 标志集(§4.1)，超类与 C 在不同的运行时包中(§5.3)。
  - 超类的 PermittedSubciasses 属性的类数组中没有条目引用名称为 N 的类或接口。
- 否则，如果 C 是一个类，而在 C 中声明的某个实例方法可以重写(§5.4.5)在 C 的超类中声明的 final 实例方法，派生就会抛出 IncompatibleClassChangeError。

4. 如果 C 有任何直接超接口，C 对其直接超接口的符号引用将使用§5.4.3.1 的算法进行解析。

因类或接口解析失败而引发的任何异常都可以因派生而引发。此外，派生必须检测以

下问题:

- 如果 C 的任何超接口是 C 本身，派生就会抛出 ClassCircularityError。
- 否则，如果任何被命名为 C 的直接超接口的类或接口实际上不是接口，派生就会抛出 IncompatibleClassChangeError。
- 否则，对于每个由 C 命名的直接超接口，如果这个超级接口有一个 PermittedSubclasses 属性(§4.7.31)，并且下面任何一个都是 true，派生会抛出一个 IncompatibleClassChangeError:
  - 超接口位于与 C 不同的运行时模块中。
  - C 没有设置 acc_public 标志集(§4.1)，超接口与 C 在不同的运行时包中。
  - 超接口的 PermittedSubclasses 属性的类数组中没有条目引用名称为 N 的类或接口。

如果在步骤 1-4 中没有抛出异常，则类或接口 C 的派生成功。Java 虚拟机将 C 标记为 L 作为它的定义加载器，记录 L 是 C 的启动加载器(§5.3.4)，并在方法区(§2.5.4)创建 C。

> 当派生成功时，加载和创建 C 的过程直到加载 C 所涉及的每个类加载器(直接或间接地)返回 C 作为结果时才结束。根据用户定义的类加载器的行为，加载和创建 C 的过程可能会失败(§5.3.2)。

如果在步骤 1-4 中抛出异常，则类或接口 C 的派生会失败，并产生该异常。

5.3.6 模块和层

Java 虚拟机支持将类和接口组织成模块。模块 M 中的类或接口 C 的成员关系用来控制 M 以外模块中的类和接口对 C 的访问(§5.4.4)。

模块成员关系是根据运行时包定义的(§5.3)。程序决定每个模块中的包的名称，以及将创建命名包的类和接口的类加载器;然后，它将包和类加载器指定给类 ModuleLayer 的 defineModules 方法的调用。调用 defineModules 会导致 Java 虚拟机创建与类加载器的运行时包相关联的新运行时模块。

每个运行时模块都指示它导出的运行时包，这将影响对这些运行时包中的公共类和接口的访问。每个运行时模块还指示它读取的其他运行时模块，这将影响其自身代码对这些运行时模块中的公共类型和接口的访问。

我们说一个类在运行时模块中，当且仅当类的运行时包与该运行时模块相关联(或者将会相关联，如果实际创建了类)。

由类加载器创建的类恰好在一个运行时包中，因此也恰好在一个运行时模块中，因为 Java 虚拟机不支持一个运行时包与多个运行时模块关联(或者更形象地说，“拆分”)。

运行时模块通过 defineModules 的语义隐式地绑定到一个类加载器。另一方面，类加载器可以在多个运行时模块中创建类，因为 Java 虚拟机不要求类加载器的所有运行时包与同一个运行时模块相关联。

> 换句话说，类加载器和运行时模块之间的关系不需要是 1:1。对于要加载的一组给定模块，如果程序可以确定每个模块中的包的名称只在该模块中找到，那么程序可以只指定一个类加载器来调用 defineModules。这个类加载器将跨多个运行时模块创建类。

每个由 defineModules 创建的运行时模块都是一个层的一部分。一层表示一组类加载器，它们共同用于在一组运行时模块中创建类。有两种层:Java 虚拟机提供的引导层和用户定义的层。引导层是在 Java 虚拟机启动时以一种依赖于实现的方式创建的。它关联标准运行时模块 java.base 和使用引导类加载器定义的标准运行时包，例如 java.lang。用户定义层是由程序创建的，目的是构造依赖于 java.base 的运行时模块集和其他标准运行时模块。

根据 defineModules 的语义，运行时模块隐式地是一个层的一部分。但是，类加载器可以在不同层的运行时模块中创建类，因为同一个类加载器可以被多个 defineModules 调用指定。访问控制由类的运行时模块控制，而不是由创建类的类加载器或类加载器服务的层控制。

为层指定的一组类加载器，以及作为层一部分的一组运行时模块，在层创建后是不可变的。但是，ModuleLayer 类为程序提供了对用户定义层中运行时模块之间关系的一定程度的动态控制。

如果用户定义的层包含多个类加载器，那么类加载器之间的任何委托都是创建该层的程序的责任。Java 虚拟机不会检查层的类加载器之间的委托是否与层的运行时模块之间的相互读取方式一致。此外，如果通过 ModuleLayer 类修改了层的运行时模块以读取额外的运行时模块，那么 Java 虚拟机不会检查层的类加载器是否被一些带外机制修改，从而以相应的方式委托。

> 类加载器和层之间既有相似之处，也有不同之处。一方面，层类似于类加载器，因为它们都可以分别委托给一个或多个父层或类加载器，这些层或加载器在较早的时候分别创建模块或类。也就是说，指定到一个层的模块集可能依赖于未指定到该层的模块，而不是之前指定给一个或多个父层的模块。另一方面，一个层只能用于创建一次新模块，而类加载器可以通过多次调用 defineClass 方法来在任何时候创建新的类或接口。

类加载器可以在运行时包中定义一个类或接口，而这个类加载器服务的任何层都不与运行时模块关联。如果运行时包包含一个未被 defineModules 指定的命名包，或者如果类或接口有一个简单的二进制名称(§4.2.1)，因此是包含一个未命名包的运行时包的成员(JLS§7.4.2)，就可能发生这种情况。在这两种情况下，类或接口都被视为隐式绑定到类加载器的特殊运行时模块的成员。这个特殊的运行时模块被称为类加载器的未命名模块。类或接口的运行时包与类加载器的未命名模块相关联。对于未命名的模块有特殊的规则，旨在最大限度地与其他运行时模块进行互操作，如下所示:

- 类加载器的未命名模块与绑定到同一类加载器的所有其他运行时模块不同。
- 类加载器的未命名模块不同于绑定到其他类加载器的所有运行时模块(包括未命名模块)。
- 每个未命名模块读取每个运行时模块。
- 每个未命名的模块导出到每个运行时模块，每个与自身关联的运行时包。

5.4 链接

链接类或接口需要验证和准备类或接口、它的直接超类、它的直接超接口以及它的元素类型(如果它是数组类型)。链接还涉及解析类或接口中的符号引用，尽管不一定在验证和准备类或接口的同时。

该规范允许在发生链接活动(以及由于递归而发生的加载)时的实现灵活性，前提是保持以下所有属性:

- 类或接口在链接之前已经完全加载。
- 类或接口在初始化之前是完全验证和准备的。
- 在链接期间检测到的错误会在程序中的某个点上抛出，在该点上，程序采取的某些操作可能直接或间接地需要链接到涉及错误的类或接口。
- 对动态计算的常量的符号引用也不会被解析直到 (i) 执行指向它的 ldc、ldc_w 或 ldc2_w 指令, 或者(ii) 调用将其作为静态参数引用的 bootstrap 方法。

直到调用将其作为静态参数引用的 bootstrap 方法时，才解析对动态计算的调用站点的符号引用。

例如，Java 虚拟机实现可能会选择“惰性”链接策略，其中类或接口中的每个符号引用(上面的符号引用除外)在使用时都是单独解析的。另外，实现也可以选择“即时”链接策略，即在验证类或接口时，立即解析所有符号引用。这意味着在某些实现中，在类或接口初始化之后，解析过程可能会继续。无论采用哪种策略，解析期间检测到的任何错误都必须在程序中(直接或间接)使用对类或接口的符号引用的某个点抛出。

因为链接涉及到新数据结构的分配，所以它可能会失败，导致 OutOfMemoryError 错误。

5.4.1 验证

验证(§4.10)确保类或接口的二进制表示在结构上是正确的(§4.9)。验证可能会导致加载额外的类和接口(§5.3)，但不需要对它们进行验证或准备。

如果类或接口的二进制表示不满足§4.9 中列出的静态或结构约束，则必须在程序中导致类或接口被验证的点抛出 VerifyError。

如果 Java 虚拟机验证类或接口的尝试失败，因为抛出了一个错误，该错误是 LinkageError(或子类)的实例，那么后续验证类或接口的尝试总是失败，并且抛出了与初始验证尝试相同的错误。

5.4.2 准备

准备工作包括为类或接口创建静态字段，并将这些字段初始化为默认值(§2.3，§2.4)。这并不需要执行任何 Java 虚拟机代码;静态字段的显式初始化是作为初始化(§5.5)的一部分执行的，而不是准备。

在准备类或接口 C 的过程中，Java 虚拟机也强加了加载约束(§5.3.4):

1. 设 L_1 是 C 的定义加载器。对于每一个在 C 中声明的实例方法 m，它可以重写(§5.4.5)在超类或超接口<D, L_2>中声明的实例方法，Java 虚拟机强加了如下的加载约束。

   假设 m 的返回类型为 T_r, m 的形参类型为 T_{f1}，…T_{fn} 的:

   如果 T_r 不是数组类型，则令 T_0 为 T_r;否则，设 T_0 为 T_r 的元素类型。

   对 i = 1 到 n: 如果 T_{fi} 不是数组类型, 则令 T_i 为 T_{fi};否则, 设 T_i 为 T_{fi} 的元素类型。

   Then $Ti^{L1} = Ti^{L2}$ for i = 0 to n。

2. 对于在 C 的超接口<I，L_3>中声明的每个实例方法 m，如果 C 本身没有声明可以重写 m 的实例方法，则针对 C 和<I，L_3>中的方法 m 选择一个方法（§5.4.6）。设 < D，L_2 > 是声明所选方法的类或接口。Java 虚拟机施加如下加载约束。

   假设 m 的返回类型为 T_r，并且 m 的形式参数类型为 T_{f1}, ..., T_{fn}:

   如果 T_r 不是数组类型，则设 T_0 为 T_r；否则，设 T_0 为 T_r 的元素类型。

   对 i = 1 到 n: 如果 T_{fi} 不是数组类型, 则设 T_i 为 T_{fi};否则, 设 T_i 为 T_{fi} 的元素类型。

   那么 $Ti^{L2} = Ti^{L3}$ 对 i = 0 到 n。

准备工作可以在创建之后的任何时候进行，但必须在初始化之前完成。

5.4.3 解析

许多 Java 虚拟机指令- anewarray, checkcast, getfield, getstatic, instanceof, invokedynamic, invokeinterface, invokespecial, invokestatic, invokevirtual, ldc, ldc_w, ldc2_w, multianewarray, new, putfield, 和 putstatic -依赖于运行时常量池中的符号引用。执行任何这些指令都需要解析符号引用。

解析是从运行时常量池中的符号引用动态确定一个或多个具体值的过程。最初，运行时常量池中的所有符号引用都未解析。

对(i)类或接口，(ii)字段，(iii)方法，(iv)方法类型，(v)方法句柄，或(vi)动态计算的常量的未解析符号引用的解析按照§5.4.3.1 至§5.4.3.5 给出的规则进行。在前三节中，符号引用出现在其运行时常量池中的类或接口被标记为 D。则:

- 如果在解析符号引用期间没有发生错误，则解析成功。

后续解析符号引用的尝试总是会成功，并且会产生初始解析产生的相同实体。如果符号引用指向一个动态计算的常量，则后续尝试不会重新执行 bootstrap 方法。

- 如果在解析符号引用时发生错误，那么它要么是(i) IncompatibleClassChangeError 的实例(或子类);(ii)由于解析或调用 bootstrap 方法而产生的 Error (或子类)实例;或(iii)由于类加载失败或违反加载器约束而产生的 LinkageError(或子类)的实例。必须在程序中(直

接或间接)使用符号引用的地方抛出错误。

解析符号引用的后续尝试总是失败，并出现初始解析尝试引发的相同错误。如果符号引用指向一个动态计算的常量，则后续的尝试不会重新执行 bootstrap 方法。

> 由于最初尝试解析时发生的错误会在随后的尝试中再次抛出，因此一个模块中的类试图通过解析其运行时常量池中的符号引用访问不同模块中未报告的公共类型时，将始终收到指示不可访问类型的相同错误（§5.4.4），即使 JavaSE 平台 API 用于在类首次尝试后的某个时间动态导出公共类型的包。

对动态计算的调用站点的未解析符号引用的解析按照§5.4.3.6 给出的规则进行。则:

- 如果在解析符号引用期间没有发生错误，那么只有 class 文件中需要解析的指令才能成功解析。这个指令必须有一个 invokedynamic 的操作码。

  在 class 文件中通过该指令解析符号引用的后续尝试总是很容易成功，并导致初始解析产生的相同实体。对于这些后续尝试，不会重新执行引导方法。

  对于任何操作码的 class 文件中的所有其他指令(它们在运行时常量池中指示与上面的 invokedynamic 指令相同的条目)，符号引用仍然未解析。

- 如果在解析符号引用时发生错误，那么它要么是(i) IncompatibleClassChangeError(或子类)的实例;(ii)由于解析或调用 bootstrap 方法而产生的 Error (或子类)实例;或(iii)由于类加载失败或违反加载约束而产生的 LinkageError(或子类)的实例。必须在程序中(直接或间接)使用符号引用的地方抛出错误。

  class 文件中相同指令解析符号引用的后续尝试总是失败，并产生初始解析尝试所引发的相同错误。对于这些后续尝试，不会重新执行引导方法。

  对于任何操作码的 class 文件中的所有其他指令(它们在运行时常量池中指示与上面的 invokedynamic 指令相同的条目)，符号引用仍然未解析。

  上面的某些指令在解析符号引用时需要进行额外的链接检查。例如，为了让 getfield 指令成功解析它所操作的字段的符号引用，它不仅必须完成§5.4.3.2 中给出的字段解析步骤，还必须检查字段是否不是静态的。如果是静态字段，则必须抛出链接异常。

  特定于特定 Java 虚拟机指令执行的检查所生成的链接异常在该指令的描述中给出，在本解决方案的一般讨论中不涉及。请注意，这种异常虽然被描述为 Java 虚拟机指令执行的一部分，而不是解析的一部分，但仍然被正确地视为解析失败。

5.4.3.1 类和接口解析

要将未解析的符号引用从 D 解析到由 N 表示的类或接口 C，请执行以下步骤:

1. D 的定义加载器用于加载并由此创建由 N 表示的类或接口。这个类或接口是 C。过程的详细信息在§5.3 中给出。

   因此，由于加载失败从而创建 C 而引发的任何异常都可能由于类和接口解析失败而引

发。

2. 如果 C 是数组类，其元素类型是引用类型，则通过递归调用§5.4.3.1 中的算法来解析对表示元素类型的类或接口的符号引用。

3. 最后，对从 D 到 C 的访问进行访问控制(§5.4.4)。

如果步骤 1 和 2 成功，但步骤 3 失败，则 C 仍然有效和可用。然而，解析失败，D 被禁止访问 C。

5.4.3.2 字段解析

要解析类或接口 C 中从 D 到字段的未解析的符号引用，必须先解析字段引用给出的对 C 的符号引用(§5.4.3.1)。因此，由于类或接口引用解析失败而引发的任何异常都可能由于字段解析失败而引发。如果对 C 的引用可以成功解析，则会引发与字段引用本身解析失败相关的异常。

解析字段引用时，字段解析首先尝试查找 C 中的引用字段及其超类:

1. 如果 C 声明的字段具有字段引用指定的名称和描述符，则字段查找成功。声明的字段是字段查找的结果。

2. 否则，字段查找将递归应用于指定类或接口 C 的直接超接口。

3. 否则，如果 C 有一个超类 S，字段查找将递归应用到 S。

4. 否则，字段查找将失败。

然后确定字段解析的结果:

- 如果字段查找失败，字段解析将抛出 NoSuchFieldError。
- 否则，字段查找成功。D 对字段进行访问控制，该字段是字段查找的结果(§5.4.4)。则:
  - 如果访问控制失败，字段解析也会因为同样的原因失败。
  - 否则，访问控制成功。施加加载约束，如下所示。

    设 < E，L_1 > 为实际声明引用字段的类或接口。设 L_2 是 D 的定义加载器。假设引用字段的类型为 T_f：如果 T_f 不是数组类型，则 T 为 T_f；否则，设 T 为 T_f 的元素类型。

    Java 虚拟机施加 $T^{L1}=T^{L2}$ 的加载约束。

    如果施加该约束导致违反任何加载约束（§5.3.4），则字段解析失败。否则，字段解析成功。

5.4.3.3 方法解析

要解析类 C 中从 D 到方法的未解析的符号引用，首先要解析由方法引用给出的对 C 的符号引用(§5.4.3.1)。因此，因类引用解析失败而引发的任何异常都可能因方法解析失败而引发。如果对 C 的引用能够成功解析，那么就会引发与方法引用本身解析相关的异常。

解析方法引用时:

1. 如果 C 是一个接口，方法解析会抛出一个 IncompatibleClassChangeError。
2. 否则，方法解析会尝试在 C 及其超类中定位被引用的方法:
   - 如果 C 用方法引用指定的名称声明了一个方法，并且声明是签名多态方法(§2.9.3)，那么方法查找成功。描述符中提到的所有类名都被解析(§5.4.3.1)。

     解析的方法是签名多态方法声明。C 没有必要声明一个方法，该方法的描述符由方法引用指定。
   - 否则，如果 C 声明的方法具有方法引用指定的名称和描述符，则方法查找成功。
   - 否则，如果 C 有一个超类，方法解析的第 2 步将在 C 的直接超类上递归调用。
3. 否则，方法解析尝试在指定类 C 的超接口中定位被引用的方法:
   - 如果 C 中由方法引用指定的名称和描述符的最大特定超接口方法恰好包括一个没有设置 ACC_ABSTRACT 标志集的方法，那么这个方法将被选中，并且方法查找成功。
   - 否则，如果 C 的任何超接口声明了一个方法，该方法的名称和描述符由方法引用指定，并且既没有设置其 ACC_PRIVATE 标志，也没有设置 ACC_STATIC 标志，则任意选择其中一个，方法查找成功。
   - 否则，方法查找将失败。

对于特定的方法名和描述符，类或接口 C 的最大特定超接口方法是以下所有条件都为 true 的任何方法:

- 该方法在 C 的超接口(直接或间接)中声明。
- 该方法使用指定的名称和描述符声明。
- 该方法既没有设置 ACC_PRIVATE 标志，也没有设置 ACC_STATIC 标志。
- 在接口 I 中声明方法的地方，不存在其他在接口 I 的子接口中声明的具有指定名称和描述符的最大特定的 C 超接口方法。

方法解析结果确定如下:

- 如果方法查找失败，方法解析将抛出一个 NoSuchMethodError。
- 否则，方法查找成功。访问控制应用于从 D 到方法查找的结果(§5.4.4)的方法访问。则:
  - 如果访问控制失败，方法解析也会因为同样的原因失败。
  - 否则，访问控制成功。施加加载约束，如下所示。

    设 < E，L_1 > 是被引用方法 m 实际声明的类或接口。设 L_2 是 D 的定义加载器。

假设 m 的返回类型为 T_r, m 的形参类型为 T_{f1}, ..., T_{fn}:

如果 T_r 不是数组类型，则令 T_0 为 T_r;否则，设 T_0 为 T_r 的元素类型。

对 i = 1 到 n: 如果 T_{fi} 不是数组类型，则让 T_i 为 T_{fi};否则，设 T_i 为 T_{fi} 的元素类型。

Java 虚拟机强加了加载约束 Ti^{L1} = Ti^{L2} 对 i = 0 到 n。

如果施加这些约束导致任何加载约束被违反(§5.3.4)，那么方法解析失败。否则，方法解析成功。

当解析在类的超接口中搜索方法时，最好的结果是识别一个最大特定的非抽象方法。这个方法可能会通过方法选择来选择，所以最好为它添加类加载器约束。

否则，结果是不确定的。这并不新鲜:Java 虚拟机规范从来没有明确指出选择了哪种方法，以及应该如何打破“联系”。在 JavaSE8 之前，这主要是一个不可观察的区别。然而，从 JavaSE8 开始，接口方法集更加异构，因此必须小心避免不确定性行为的问题。因此:

- 解析将忽略私有和静态的超接口方法。这与 Java 编程语言一致，在 Java 编程语言中，此类接口方法不被继承。
- 由已解析方法控制的任何行为都不应依赖于该方法是否抽象。

注意，如果解析的结果是一个抽象方法，则引用的类 C 可能是非抽象的。要求 C 是抽象的会与超接口方法的不确定性选择相冲突。相反，解析假定被调用对象的运行时类具有该方法的具体实现。

5.4.3.4 接口方法解析

要解析从 D 到接口 C 中接口方法的未解析的符号引用，首先要解析接口方法引用给出的对 C 的符号引用(§5.4.3.1)。因此，任何可能因接口引用解析失败而引发的异常都可能因接口方法解析失败而引发。如果对 C 的引用能够成功解析，则会引发与解析接口方法引用本身相关的异常。

解析接口方法引用时:

1. 如果 C 不是接口，则接口方法解析会抛出 IncompatibleClassChangeError。
2. 否则，如果 C 声明的方法具有接口方法引用指定的名称和描述符，则方法查找成功。
3. 否则，如果类 Object 声明了一个方法，该方法的名称和描述符由接口方法引用指定，该方法设置了 ACC_PUBLIC 标志而没有设置 ACC_STATIC 标志，则方法查找成功。
4. 否则，如果 C 中最大特定的超接口方法(§5.4.3.3)中方法引用指定的名称和描述符包含一个没有设置 ACC_ABSTRACT 标志集的方法，那么这个方法将被选中，并且方法查找成功。
5. 否则，如果 C 的任何超接口声明了一个方法，方法引用指定的名称和描述符既没有设置 ACC_PRIVATE 标志也没有设置 ACC_STATIC 标志，则任意选择其中一个方法，方法查找成功。
6. 否则，方法查找将失败。

接口方法解析的结果确定如下:

- 如果方法查找失败，接口方法解析将抛出一个 NoSuchMethodError。
- 否则，方法查找成功。访问控制应用于从 D 到方法查找的结果(§5.4.4)的方法的访问。则:
  - 如果访问控制失败，接口方法解析也会因为同样的原因失败。
  - 否则，访问控制成功。施加的加载约束如下所示。

    设 < E，L_1 > 为实际声明引用接口方法 m 的类或接口。设 L_2 是 D 的定义加载器。假设 m 的返回类型为 T_r，并且 m 的形式参数类型为 T_{f1}, ..., T_{fn}:

    如果 T_r 不是数组类型，则设 T_0 为 T_r；否则，设 T_0 为 T_r 的元素类型。

    对 i = 1 到 n: 如果 T_{fi} 不是数组类型，则设 T_i 为 T_{fi}；否则，设 T_i 为 T_{fi} 的元素类型。

    Java 虚拟机施加了加载约束 $Ti^{L1} = Ti^{L2}$ 对 i = 0 到 n。

    如果强加这些约束导致任何加载约束被违反(§5.3.4)，那么接口方法解析失败。否则，接口方法解析成功。

    > 访问控制是必要的，因为接口方法解析可能会选择接口 C 的私有方法。(在 Java SE 8 之前，接口方法解析的结果可以是 Object 类的非公共方法，也可以是 Object 类的静态方法;这样的结果与 Java 编程语言的继承模型不一致，并且在 Java SE 8 及以上版本中是不允许的。

5.4.3.5 方法类型和方法句柄解析

要解析对方法类型未解析的符号引用，就好像解析了对类和接口(§5.4.3.1)未解析的符号引用，这些类和接口的名称对应于方法描述符中给出的类型(§4.3.3)。

因此，类引用解析失败可能引发的任何异常都可能因为方法类型解析失败而引发。

成功的方法类型解析的结果是对 java.lang.invoke.MethodType 实例的引用，该实例表示方法描述符。

> 无论运行时常量池是否实际包含对方法描述符中指出的类和接口的符号引用，都会发生方法类型解析。此外，解析被认为发生在未解析的符号引用上，因此，如果稍后可以加载合适的类和接口，解析一种方法类型的失败不一定会导致使用相同的文本方法描述符解析另一种方法类型的失败。

对方法句柄的未解析符号引用的解析更为复杂。Java 虚拟机解析的每个方法句柄都有一个等效的指令序列，称为其字节码行为，由方法句柄的类型表示。表 5.4.3.5-A 给出了 9 种方法句柄的整数值和描述。

指令序列对字段或方法的符号引用用 C.x:T 表示，其中 x 和 T 是字段或方法的名称和描述符(§4.3.2，§4.3.3)，C 是可以找到字段或方法的类或接口。

表 5.4.3.5-A. 方法句柄的字节码行为

种类	描述	说明

1	REF_getField	getfield C.f:T
2	REF_getStatic	getstatic C.f:T
3	REF_putField	putfield C.f:T
4	REF_putStatic	putstatic C.f:T
5	REF_invokeVirtual	invokevirtual C.m:(A*)T
6	REF_invokeStatic	invokestatic C.m:(A*)T
7	REF_invokeSpecial	invokespecial C.m:(A*)T
8	REF_newInvokeSpecial	new C; dup; invokespecial C.<init>:(A*)V
9	REF_invokeInterface	invokeinterface C.m:(A*)T

让 MH 成为正在解析的方法句柄(§5.1)的符号引用。另外:

- 设 R 是 MH 中包含的字段或方法的符号引用。

  R 派生自 CONSTANT_MethodHandle 的 reference_index 项引用的 CONSTANT_Fieldref, CONSTANT_Methodref 或 CONSTANT_InterfaceMethodref 结构，MH 派生自 CONSTANT_MethodHandle。

  > 例如，R 是对 C 的符号引用。f 表示字节码行为类型 1，以及对 C 的符号引用。<init>表示字节码行为类型为 8。

  如果 MH 的字节码行为是类型 7 (REF_invokeSpecial)，那么 C 必须是当前类或接口，当前类的超类，当前类或接口的直接超接口，或 Object。

- 设 T 是 R 引用的字段的类型，或者 R 引用的方法的返回类型。设 A*是 R 引用的方法的参数类型的序列(可能为空)。

  T 和 A*派生自 Constant_NameAndType 结构，由从中派生 R 的 Constant_Fieldref、Constant_Methodref 或 Constant_InterfaceMethodref 结构中的 name_and_type_index 项引用。

要解析 MH，将使用以下四个步骤解析 MH 字节码行为中对类、接口、字段和方法的所有符号引用:

1. R 被解析。当 MH 的字节码行为是种类 1、2、3 或 4 时，就好像通过字段解析(§5.4.3.2)发生;当 MH 的字节码行为是种类 5、6、7 或 8 时，就好像通过方法解析(§5.4.3.3)发生;当 MH 的字节码行为是种类 9 时，就好像通过接口方法解析(§5.4.3.4)发生。

2. 以下约束适用于解析 R 的结果。这些约束对应于那些将在验证或执行相关字节码行为的指令序列期间强制执行的约束。

   - 如果 MH 的字节码行为是种类 8 (REF_newInvokeSpecial)，那么 R 必须解析为类 C 中声明的实例初始化方法。

- 如果 R 解析为受保护的成员，则根据 MH 的字节码行为类型应用以下规则：
  - 对于种类 1, 3, 和 5 (REF_getFieid, REF_putFieid, 和 REF_invokeVirtual): 如果 C.f 或 C.m 解析为受保护的字段或方法，并且 C 位于与当前类不同的运行时包中，则 C 必须可赋值给当前类。
  - 对于种类 8(REF_newinvokeSpeciai): 如果 C. < init > 解析为受保护的方法，则必须在与当前类相同的运行时包中声明 C。
- R 必须根据 MH 的字节码行为的类型解析为静态或非静态成员：
  - 对于种类 1, 3, 5, 7, 和 9 (REF_getField, REF_putField, REF_invokeVirtual, REF_invokeSpecial, 和 REF_invokeInterface): C.f 或 C.m 必须解析为非静态字段或方法。
  - 对于种类 2, 4, 和 6 (REF_getstatic, REF_putstatic, 和 REF_invokeStatic): C.f 或 C.m 必须解析为静态字段或方法。

3. 解析就像是对类和接口的未解析的符号引用，这些类和接口的名称对应于 A*中的每种类型，并按此顺序对应于类型 T。

4. 获得对 java.lang.invoke.MethodType 实例的引用时，就好像通过解析未解析的符号引用来获得方法类型，该方法类型包含表 5.4.3.5-B 中为 MH 类型指定的方法描述符。

   就好像对方法句柄的符号引用包含了对已解析方法句柄最终将拥有的方法类型的符号引用。方法类型的详细结构通过检查表 5.4.3.5-B 得到。

表 5.4.3.5-B. 方法句柄的方法描述符

种类	说明	方法描述符
1	REF_getField	(C)T
2	REF_getStatic	()T
3	REF_putField	(C,T)V
4	REF_putStatic	(T)V
5	REF_invokeVirtual	(C,A*)T
6	REF_invokeStatic	(A*)T
7	REF_invokeSpecial	(C,A*)T
8	REF_newInvokeSpecial	(A*)C
9	REF_invokeInterface	(C,A*)T

在步骤 1、3 和 4 中，由于解析类、接口、字段或方法的符号引用失败而引发的任何异常都可能由于解析方法句柄失败而引发。在第 2 步中，由于指定的约束导致的任何失败都会导致由于 IllegalAccessError 导致的方法句柄解析失败。

其目的是，解析方法句柄的环境与 Java 虚拟机在字节码行为中成功验证和解析符号引用的

环境完全相同。特别是，可以在相应的正常访问是合法的类中创建私有、受保护和静态成员的方法句柄。

方法句柄解析成功的结果是对 java.lang.invoke.MethodHandle 实例的引用，该实例表示方法句柄 MH。

此 java.lang.invoke.MethodHandle 实例的类型描述符是上述方法句柄解析的第三步中生成的 java.lang.invoke.MethodType 实例。

方法句柄的类型描述符是这样的，在方法句柄上对 java.lang.invoke.MethodHandle 中的 invokeExact 的有效调用具有与字节码行为完全相同的栈效果。对一组有效的参数调用此方法句柄具有完全相同的效果，并返回与相应字节码行为相同的结果(如果有的话)。

如果 R 引用的方法设置了 ACC_VARARGS 标志(§4.6)，那么 java.lang.invoke.MethodHandle 实例就是一个可变参数方法句柄;否则，它是一个固定参数方法句柄。

可变参数方法句柄在通过 invoke 调用时执行参数列表装箱(JLS§15.12.4.2)，而它相对于 invokeExact 的行为就像没有设置 ACC_VARARGS 标志一样。

如果 R 引用的方法设置了 ACC_VARARGS 标志，且 A*是空序列或 A*中的最后一个参数类型不是数组类型，则方法句柄解析抛出 IncompatibleClassChangeError。也就是说，可变参数方法句柄的创建失败。

Java 虚拟机的实现并不需要保留方法类型或方法句柄。也就是说，两个结构相同的方法类型或方法句柄的不同符号引用可能不会分别解析为 java.lang.invoke.MethodType 或 java.lang.invoke.MethodHandle 的相同实例。

Java SE 平台 API 中的 java.lang.invoke.MethodHandles 类允许创建无字节码行为的方法句柄。它们的行为是由创建它们的 java.lang.invoke.MethodHandles 的方法定义的。例如，在调用方法句柄时，可能首先对其参数值应用转换，然后将转换后的值提供给另一个方法句柄的调用，然后对从该调用返回的值应用转换，然后将转换后的值作为自己的结果返回。

5.4.3.6 动态计算常量和调用站点解析

要将未解析的符号引用 R 解析到动态计算的常量或调用站点，有三个任务。首先，检查 R 以确定哪个代码将作为它的引导方法，以及哪些参数将被传递给该代码。其次，将参数打包到数组中，并调用 bootstrap 方法。第三，对 bootstrap 方法的结果进行验证，并将其作为解析的结果。

第一个任务包括以下步骤:

1. R 给出了对引导方法句柄的符号引用。bootstrap 方法句柄被解析(§5.4.3.5)以获得 java.lang.invoke.MethodHandle 实例的引用。

   任何由于解析方法句柄的符号引用失败而引发的异常都可以在此步骤中引发。

   如果 R 是对动态计算的常量的符号引用，那么就让 D 作为引导方法句柄的类型描述符。(也就是说，D 是对 java.lang.invoke.MethodType 实例的引用。)D 指出的第一个参数类

型必须是 java.lang.invoke.MethodHandles.Lookup，否则解析会因 BootstrapMethodError 而失败。由于历史原因，动态计算调用站点的引导方法句柄没有类似的约束。

2. 如果 R 是对动态计算的常量的符号引用，那么它给出一个字段描述符。

   如果字段描述符指示一个原生类型，则获得对表示该类型的预定义 Class 对象的引用(请参阅类 Class 中的方法 isPrimitive)。

   否则，字段描述符指示类或接口类型或数组类型。获得表示字段描述符所指示类型的 Class 对象的引用，就好像通过解析未解析的类或接口符号引用（§5.4.3.1），其名称对应于字段描述符所示类型。

   由于未能解析类或接口的符号引用而引发的任何异常都可以在此步骤中引发。

3. 如果 R 是对动态计算的调用站点的符号引用，则它给出一个方法描述符。

   获得了对 java.lang.invoke.MethodType 的实例的引用，就像通过解析未解析的对方法类型(§5.4.3.5)的符号引用一样，该方法类型具有与方法描述符相同的参数和返回类型。

   任何由于解析方法类型的符号引用失败而引发的异常都可以在此步骤中引发。

4. R 给出零个或多个静态参数，它们将特定于应用程序的元数据传递给 bootstrap 方法。每个静态参数 A 按 R 给出的顺序被解析如下:

   - 如果 A 是一个字符串常量，则获得对类 String 的实例的引用。
   - 如果 A 是一个数值常量，则通过以下过程获得对 java.lang.invoke.MethodHandle 实例的引用:

     a) 设 v 为数值常量的值，设 T 为对应于数值常量类型的字段描述符。

     b) 假设 MH 是一个方法句柄，就像通过调用 java.lang.invoke.MethodHandles 的身份方法产生的一样，该方法句柄带有一个表示类 Object 的参数。

     c) 通过调用带有方法描述符(T)LJava/lang/Object 的 MH.invoke(v)来获得对 java.lang.invoke.MethodHandle 实例的引用。

   - 如果 A 是对动态计算的常量的符号引用，其字段描述符指示原生类型 T，则解析 A，生成原生值 v。给定 v 和 T，根据上面为数值常量指定的过程，获得对 java.lang.invoke.MethodHandle 实例的引用。
   - 如果 A 是任何其他类型的符号引用，那么结果就是解析 A 的结果。

   在运行时常量池中的符号引用中，动态计算常量的符号引用是特殊的，因为它们来源于 constant_pool 条目，这些条目可以通过 BootstrapMethods 属性从语法上引用自己(§4.7.23)。但是，Java 虚拟机不支持将符号引用解析为依赖于自身的动态计算常量(也就是说，作为其自身引导方法的静态参数)。相应地，当 R 和 A 都是对动态计算常量的

符号引用时，如果 A 与 R 相同，或者 A 给出一个静态参数(直接或间接)引用 R，则解析失败，在需要重新解析 R 的地方出现 StackOverflowError。

> 不像类初始化(§5.5)，在未初始化的类之间允许循环，解析不允许在动态计算的常量的符号引用中使用循环。如果解析的实现递归使用栈，则自然会发生 StackOverflowError。如果不是，则需要实现检测循环，而不是无限循环或为动态计算的常量返回默认值。
>
> 如果引导方法的主体引用当前正在解析的动态计算常量，则可能出现类似的循环。这对于 invokedynamic 引导始终是可能的，并且不需要在解析中进行特殊处理；递归调用 invokeWithArguments 自然会导致 StackOverflowError。

由于符号引用解析失败而引发的任何异常都可以在此步骤中引发。

第二个任务，调用引导方法句柄，涉及以下步骤:

1. 数组的组件类型为 Object，长度为 n+3，其中 n 是由 R (n > 0)给出的静态参数的数量。

   数组的第 0 个组件被设置为对出现 R 的类的 java.lang.invoke.MethodHandles.Lookup 实例的引用，就像通过调用 java.lang.invoke.MethodHandles 的 lookup 方法生成的一样。

   数组的第一个组件被设置为对表示 N 的字符串实例的引用，N 是由 R 给出的非限定名称。

   数组的第二个组件被设置为对 Class 或 java.lang.invoke.MethodType 实例的引用，该实例是先前为 R 给出的字段描述符或方法描述符获得的。

   数组的后续组件被设置为先前从解析 R 的静态参数(如果有的话)中获得的引用。引用在数组中出现的顺序与对应的静态参数由 R 给出的顺序相同。

   > Java 虚拟机实现可以跳过数组的分配，并且在不改变任何可观察行为的情况下，直接将参数传递给引导方法。

2. 引导方法句柄被调用，就像调用 BMH.invokeWithArguments(args)一样，其中 BMH 是引导方法句柄，args 是上面分配的数组。

   > 由于 java.lang.invoke.MethodHandle 的 invokeWithArguments 方法的行为，引导方法句柄的类型描述符不需要与参数的运行时类型完全匹配。例如，引导方法句柄的第二个参数类型(对应于上面数组的第一个组件中给出的非限定名称)可以是 Object 而不是 String。如果引导方法句柄是可变的，则可以将部分或所有参数收集到尾随数组参数中。

   调用发生在试图解析此符号引用的线程中。如果存在几个这样的线程，则可以并发地调用引导方法句柄。访问全局应用程序数据的引导方法应该采取通常针对竞争条件的预防措施。

   如果调用因抛出 Error 或 Error 的子类的实例而失败，则解析将失败，并引发该异常。

   如果调用失败，抛出的异常不是 Error 的或 Error 的子类的实例，则解析失败并抛出 BootstrapMethodError，原因是引发的异常。

   如果多个线程并发地调用这个符号引用的引导方法句柄，Java 虚拟机将选择一个调用

的结果并将其安装到所有线程。允许为这个符号引用执行的任何其他引导方法完成，但它们的结果将被忽略。

第三个任务是验证通过调用引导方法句柄产生的引用 o，如下所示:

- 如果 R 是对动态计算的常量的符号引用，那么 o 将被转换为类型 T，即由 R 给出的字段描述符所指示的类型。

  o 的转换就像通过调用带有方法描述符(LJava/lang/Object；)T 的 MH.invoke(o)一样发生，其中 MH 是一个方法句柄，就像是通过调用带有表示类 Object 的参数的 java.lang.invoke.MethodHandles 的 identity 方法而产生的。

  o 的转换结果是解析的结果。

  如果转换因引发 NullPointerException 或 ClassCastException 而失败，则解析将失败，并抛出 BootstrapMethodError。

- 如果 R 是对动态计算的调用站点的符号引用，则 o 是解析的结果，如果它具有以下所有属性:

  - o 不是 null。
  - o 是 java.lang.invoke.CallSite 的实例或 java.lang.invoke.CallSite 的子类。
  - java.lang.invoke.CallSite 的类型在语义上等于 R 给出的方法描述符。

  如果 o 没有这些属性，解析将失败，并抛出 BootstrapMethodError。

上面的许多步骤“好像通过调用”某些方法来执行计算。在每种情况下，调用行为都由 invokestatic 和 invokevirtual 的规范详细给出。调用发生在线程和试图解析符号引用 R 的类中。但是，不需要在运行时常量池中出现相应的方法引用，也不需要使用特定的方法的操作数栈，并且不强制调用任何方法的 Code 属性的 max_stack 项的值。

如果多个线程同时尝试解析 R，则可以并发调用引导方法。因此，访问全局应用程序数据的引导方法必须对竞争条件采取预防措施。

5.4.4 访问控制

在解析过程中应用访问控制(§5.4.3)，以确保对类、接口、字段或方法的引用是允许的。如果指定的类、接口、字段或方法可被引用的类或接口访问，则访问控制成功。

类或接口 C 可被类或接口 D 访问，当且仅当下列情况之一为 true:

- C 是公共的，和 D 是同一个运行时模块的成员(§5.3.6)。
- C 是公共的，是与 D 不同的运行时模块的成员，C 的运行时模块由 D 的运行时模块读取，C 的运行时模块将 C 的运行时包导出到 D 的运行时模块。
- C 不是公共的，C 和 D 是同一个运行时包的成员。

如果 D 不能访问 C，则访问控制抛出一个 IllegalAccessError。否则，访问控制成功。

当且仅当满足以下任一条件时，类或接口 D 才能访问字段或方法 R:

- R 是公共的。
- R 是受保护的，并在类 C 中声明，D 要么是 C 的子类，要么是 C 本身。

  此外，如果 R 不是静态的，那么对 R 的符号引用必须包含对类 T 的符号引用，这样 T 要么是 D 的子类，要么是 D 的超类，要么是 D 本身。

  > 在 D 的验证过程中，要求即使 T 是 D 的超类，受保护字段访问或方法调用的目标引用必须是 D 的实例或 D 的子类(§4.10.1.8)。

- R 要么是受保护的，要么具有默认访问权限(即既不是公共的，也不是受保护的，也不是私有的)，并由与 D 相同的运行时包中的一个类声明。
- R 是私有的，由类或接口 C 声明，根据下面的嵌套测试，类或接口 C 属于与 D 相同的嵌套。

如果 D 无法访问 R，则访问控制抛出 IlLegalAccessError。否则，访问控制成功。

嵌套是一组允许相互访问它们的私有成员的类和接口。其中一个类或接口是嵌套主机。它使用 NestMembers 属性(§4.7.29)枚举属于嵌套的类和接口。它们中的每一个依次使用 NestHost 属性(§4.7.28)将其指定为嵌套主机。缺少 NestHost 属性的类或接口属于自己托管的嵌套;如果它也缺少一个 NestMembers 属性，那么这个嵌套就是一个只由类或接口本身组成的单例。

Java 虚拟机确定给定类或接口所属的嵌套(即类或接口指定的嵌套主机)作为访问控制的一部分，而不是在加载类或接口的时候。Java SE 平台 API 的某些方法可以在访问控制之前确定给定类或接口所属的嵌套，在这种情况下，Java 虚拟机在访问控制期间尊重该先前确定。

为了确定类或接口 C 是否属于与类或接口 D 相同的嵌套，应用嵌套测试。当且仅当嵌套测试成功时，C 和 D 属于同一个嵌套。嵌套测试如下:

- 如果 C 和 D 是相同的类或接口，则嵌套测试成功。
- 否则，执行以下步骤，顺序为:

  1. 设 H 为 D 的嵌套主机，如果 D 的嵌套主机之前已经确定。如果之前没有确定 D 的嵌套主机，则使用下面的算法确定，得到 H。
  2. 设 H'为 C 的嵌套主机，如果 C 的嵌套主机之前已经确定。如果之前没有确定 C 的嵌套主机，那么使用下面的算法确定它，得到 H'。
  3. 对 H 和 H'进行了比较。如果 H 和 H'是相同的类或接口，则嵌套测试成功。否则，嵌套测试失败。

类或接口 M 的嵌套主机确定如下:

- 如果 M 缺少 NestHost 属性，那么 M 就是它自己的嵌套主机。
- 否则，M 有一个 NestHost 属性，并且它的 host_class_index 项被用作 M 的运行时常量池的索引。该索引处的符号引用被解析(§5.4.3.1)。

  如果符号引用解析失败，那么 M 就是它自己的嵌套主机。由于类或接口解析失败而抛出的任何异常都不会被重新抛出。

  否则，符号引用解析成功。设 H 为解析的类或接口。M 的嵌套主机由以下规则确定:

  - 如果下列任何一个是 true，那么 M 就是它自己的嵌套主机:
    > H 与 M 不在同一个运行时包中。
    > H 缺少一个 NestMembers 属性。
    > H 有一个 NestMembers 属性，但是在它的 classes 数组中没有引用名称为 N 的类或接口的条目，其中 N 是 M 的名称。
  - 否则，H 是 M 的嵌套主机。

5.4.5 方法重写

一个实例方法 m_C 可以重写另一个实例方法 m_A，当且仅当以下所有条件都为 true:

- m_C 的名称和描述符与 m_A 相同。
- m_C 没有被标记为 ACC_PRIVATE。
- 以下之一为 true:
  - m_A 标记为 ACC_PUBLIC。
  - m_A 标记为 ACC_PROTECTED。
  - m_A 既不标记为 ACC_PUBLIC，也不标记为 ACC_PROTECTED，也不标记为 ACC_PRIVATE，并且(a) m_A 的声明出现在与 m_C 的声明相同的运行时包中，或者(b)如果 m_A 在类 A 中声明并且 m_C 在类 C 中声明，则存在在类 B 中声明的方法 m_B，使得 C 是 B 的子类并且 B 是 A 的子类并且 m_C 可以重写 m_B 并且 m_B 可以重写 m_A。

    最后一个案例的第（b）部分允许使用默认访问的方法进行“传递性重写”。例如，给定包 P 中的以下类声明:

    ```
    public  class  A          {       void m(){}}
    public  class  B extends A{public void m(){}}
    public  class  C extends B{       void m(){}}
    ```

    以及不同包中的以下类声明:

    ```
    public class D extends P.C { void m() {} }
    ```

    则:

    - B.m 可以重写 A.m.

- C.m 可以重写 B.m 和 A.m。
- D.m 可以重写 B.m 和 A.m, 但不能重写 C.m。

5.4.6 方法选择

在 invokeinterface 或 invokevirtual 指令的执行过程中，根据(i)栈上对象的运行时类型和(ii)先前由指令解析的方法来选择方法。针对类或接口 C 和方法 m_R 选择方法的规则如下:

1. 如果 m_R 被标记为 ACC_PRIVATE，那么它就是所选的方法。
2. 否则，所选方法由以下查找过程确定:
   - 如果 C 包含了一个实例方法 m 的声明，它可以重写 m_R(§5.4.5)，那么 m 就是被选择的方法。
   - 否则，如果 C 有一个超类，就会搜索可以重写 mR 的实例方法的声明，从 C 的直接超类开始，然后继续搜索该类的直接超类，以此类推，直到找到方法或不再存在其他超类为止。如果找到一个方法，它就是所选的方法。
   - 否则，确定 C 的最大特定超接口方法(§5.4.3.3)。如果恰好有一个方法匹配 m_R 的名称和描述符并且不是抽象的，那么它就是所选的方法。

     在此步骤中选择的任何最大特定超接口方法都可以重写 m_R;没有必要显式地检查这一点。

   虽然 C 通常是一个类，但当这些规则在准备过程中被应用时，它可能是一个接口(§5.4.2)。

5.5 初始化

类或接口的初始化包括执行其类或接口初始化方法(§2.9.2)。

类或接口 C 只能在以下情况下初始化:

- Java 虚拟机指令 new、getstatic、putstatic 或 invokestatic 的任意一个引用 C(§new、§getstatic、§putstatic、§invokestatic)的执行。

  在执行 new 指令时，要初始化的类就是该指令引用的类。

  在执行 getstatic、putstatic 或 invokestatic 指令时，要初始化的类或接口就是声明解析字段或方法的类或接口。

- java.lang.invoke.MethodHandle 实例的第一次调用，它是方法句柄解析(§5.4.3.5)的结果，用于类型 2 (REF_getStatic)， 4 (REF_putStatic)， 6 (REF_invokeStatic)，或 8 (REF_newInvokeSpecial)的方法句柄。

  这意味着当引导方法为 invokedynamic 指令(§invokedynamic)调用时，引导方法的类被初始化，作为调用站点说明符的持续解析的一部分。

- 在类库(§2.12)中调用某些反射方法，例如在类 Class 或包 java.lang.reflect 中。
- 如果 C 是一个类，则它的一个子类的初始化。

- 如果 C 是一个声明非抽象、非静态方法的接口，则直接或间接实现 C 的类的初始化。
- 它是 Java 虚拟机启动时的初始类或接口(§5.2)。

在初始化之前，必须链接类或接口，即验证、准备和可选地解析。

因为 Java 虚拟机是多线程的，类或接口的初始化需要谨慎的同步，因为其他线程可能试图同时初始化同一个类或接口。也有可能递归地请求类或接口的初始化，作为该类或接口初始化的一部分。Java 虚拟机的实现通过使用以下过程负责同步和递归初始化。它假设 Class 对象已经被验证和准备好了，并且 Class 对象包含指示以下四种情况之一的状态:

- 这个 Class 对象经过验证和准备，但没有初始化。
- 这个 Class 对象由某些特定的线程初始化。
- 这个 Class 对象已经完全初始化，可以使用了。
- 此 Class 对象处于错误状态，可能是因为尝试初始化但失败了。

对于每个类或接口 C，都有一个惟一的初始化锁 LC。从 C 到 LC 的映射由 Java 虚拟机实现自行决定。例如，LC 可以是 C 的 Class 对象，也可以是与那个 Class 对象相关联的监视器。初始化 C 的过程如下:

1. 同步 C 的初始化锁 LC。这涉及到等待，直到当前线程可以获取 LC。
2. 如果 C 的 Class 对象表明其他线程正在对 C 进行初始化，那么释放 LC 并阻塞当前线程，直到被告知正在进行的初始化已经完成，然后重复此过程。

   线程中断状态不受初始化过程执行的影响。
3. 如果 C 的 Class 对象指示当前线程正在对 C 进行初始化，那么这一定是一个递归的初始化请求。释放 LC 并正常完成。
4. 如果 C 的 Class 对象表示 C 已经初始化，则不需要进一步操作。释放 LC 并正常完成。
5. 如果 C 的 Class 对象处于错误状态，则不可能进行初始化。释放 LC 并抛出 NoClassDefFoundError。
6. 否则，记录当前线程正在初始化 C 的 Class 对象这一事实，并释放 LC。

   然后，用 ConstantValue 属性中的常量值(§4.7.2)初始化 C 的每个 final 静态字段，按照字段在 ClassFile 结构中出现的顺序。
7. 接下来，如果 C 是一个类而不是一个接口，那么就让 SC 作为它的超类，让 SI_1, ..., SI_n 是 C 的所有超接口(不管是直接的还是间接的)，它们声明了至少一个非抽象的、非静态的方法。超接口的顺序是由 C 直接实现的每个接口的超接口层次结构上的递归枚举给出的。对于 C 直接实现的每个接口 I(按照 C 的接口数组的顺序)，在返回 I 之前，枚举会在 I 的超接口上重复出现(按照 I 的接口数组的顺序)。

   对于列表[SC, SI_1, ..., SI_n]中的每个 S, 如果 S 尚未初始化，则递归地对 S 执行整个过程。

如有必要，先验证并准备 S。

如果 S 的初始化因为抛出异常而突然完成，那么获取 LC，将 C 的 Class 对象标记为错误，通知所有等待的线程，释放 LC，然后突然完成，抛出初始化 SC 导致的相同异常。

8. 接下来，通过查询 C 的定义加载器来确定是否为 C 启用断言。
9. 接下来，执行 C 的类或接口初始化方法。
10. 如果类或接口初始化方法的执行正常完成，那么获取 LC，将 C 的 Class 对象标记为完全初始化，通知所有等待的线程，释放 LC，并正常完成此过程。
11. 否则，类或接口初始化方法必须通过抛出一些异常 E 而突然完成。如果 E 的类不是 Error 或它的子类之一，则用 E 作为参数创建类 ExceptionInInitializerError 的新实例，并在下面的步骤中使用该对象代替 E。如果因为发生了 OutOfMemoryError 而无法创建 ExceptionInInitializerError 的新实例，则在下面的步骤中使用 OutOfMemoryError 对象代替 E。
12. 获取 LC，将 C 的 Class 对象标记为错误，通知所有等待的线程，释放 LC，并突然用原因 E 或上一步中确定的替换原因完成此过程。

Java 虚拟机实现可以优化这个过程，当它可以确定类的初始化已经完成时，省略第 1 步中的锁获取(和第 4/5 步中的释放)，前提是，根据 Java 内存模型，所有 happens-before 排序(JLS§17.4.5) (如果获得了锁，就会存在)在执行优化时仍然存在。

5.6 绑定本地方法实现

绑定是将用 Java 编程语言以外的语言编写并实现本地方法的函数集成到 Java 虚拟机中以便执行的过程。尽管这个过程传统上被称为链接，但规范中使用绑定这个术语是为了避免与 Java 虚拟机的类或接口的链接相混淆。

5.7 Java 虚拟机退出

当某些线程调用类 Runtime 或类 System 的 exit 方法，或类 Runtime 的 halt 方法，并且安全管理器允许 exit 或 halt 操作时，Java 虚拟机退出。

此外，JNI (Java 本地接口)规范描述了在使用 JNI 调用 API 加载和卸载 Java 虚拟机时 Java 虚拟机的终止。