(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 106462425 B
(45)授权公告日 2020.09.11

(21)申请号 201580027961.4

(22)申请日 2015.06.08

(65)同一申请的已公布的文献号
申请公布号 CN 106462425 A

(43)申请公布日 2017.02.22

(30)优先权数据
62/011,411 2014.06.12 US
14/726,155 2015.05.29 US

(85)PCT国际申请进入国家阶段日
2016.11.22

(86)PCT国际申请的申请数据
PCT/US2015/034728 2015.06.08

(87)PCT国际申请的公布数据
WO2015/191469 EN 2015.12.17

(73)专利权人 甲骨文国际公司
地址 美国加利福尼亚

(72)发明人 J·R·罗斯 B·戈茨

(74)专利代理机构 中国贸促会专利商标事务所有限公司 11038
代理人 刘凤香

(51)Int.Cl.
G06F 9/445(2018.01)
G06F 9/455(2006.01)
G06F 8/30(2018.01)
G06F 8/41(2018.01)

审查员 曹永敏

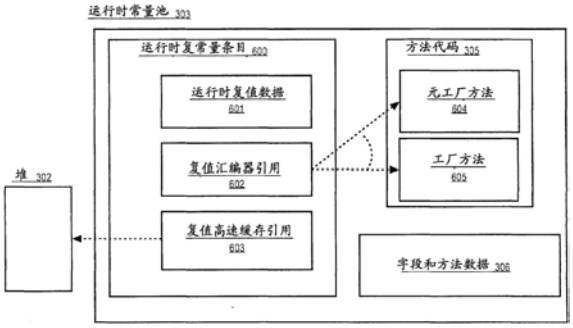
权利要求书2页 说明书27页 附图8页

(54)发明名称

使用复常量的方法和系统

(57)摘要

在方法中,虚拟机在一组指令内识别加载常量的指令;基于加载常量的指令识别数据结构中的第一条目,该数据结构识别一个或多个常量类型中的特定常量类型,其中第一条目至少指定常量数据以及用于从常量数据组装值或部分值的第一组指令;执行第一组指令以从常量数据组装值或部分值;将特定值或对特定值的引用存储到被用于在运行环境中执行的指令组之间传递值或引用的运行时数据结构上,其中特定值基于从常量数据组装的值或特定值。



CN 106462425 B

1.一种方法,包括:

在一组程序指令内识别加载常量的指令;

基于加载所述常量的所述指令在识别由程序使用的一个或多个常量的数据结构中识别与所述常量相关联的第一条目,其中所述第一条目至少指定常量数据以及用于从所述常量数据组装复值或部分复值的第一组指令,其中所述常量数据表示被所述第一组指令用来组装所述复值或所述部分复值的多个常量值,其中所述复值或所述部分复值表示所述多个常量值的聚合,其中所述数据结构包括所述多个常量值中的至少一个常量值;

执行所述第一组指令以从所述常量数据组装所述复值或所述部分复值;

将特定值或对所述特定值的引用存储到被用于在运行环境中执行的指令组之间传递值或引用的运行时数据结构上,其中所述特定值基于从所述常量数据组装的所述复值或所述部分复值,

其中所述方法由一个或多个计算设备执行。

2.如权利要求1所述的方法,其中所述常量表示数组、组、序列、字典、复数、点或对象中的一个或多个。

3.如权利要求1-2中的任何一项权利要求所述的方法,其中识别由所述程序使用的所述一个或多个常量的所述数据结构是包含多个条目的常量表,其中所述多个条目中的每个条目与不同常量相关。

4.如权利要求1-2中的任何一项权利要求所述的方法,其中执行所述第一组指令在常量值高速缓存中存储所述复值或所述部分复值。

5.如权利要求4所述的方法,其中存储在所述常量值高速缓存中的所述复值或所述部分复值被存储在其中所述复值或部分复值不可变的存储器位置处。

6.如权利要求4所述的方法,其中执行所述第一组指令生成第二组指令,所述第二组指令在被执行时,基于存储在所述常量值高速缓存中的所述复值或所述部分复值来生成所述特定值以及执行将所述特定值或对所述特定值的引用存储到所述运行时数据结构上的操作。

7.如权利要求6所述的方法,其中所述数据结构中的所述第一条目包括指向所述第一组指令的汇编器引用,并且执行所述第一组指令使得所述汇编器引用被更新为指向所述第二组指令。

8.如权利要求7所述的方法,还包括:

在第二组程序指令内识别加载所述常量的第二指令;

基于加载所述常量的所述第二指令,在识别由所述程序使用的所述一个或多个常量的所述数据结构中识别与所述常量相关联的第一条目,其中所述第一条目至少识别所述常量值高速缓存中的所述复值或部分复值和所述第二组指令;

执行所述第二组指令以基于所述常量值高速缓存中的所述复值或所述部分复值将第二特定值或对所述第二特定值的引用存储到所述运行时数据结构上。

9.如权利要求6所述的方法,其中执行所述第一组指令组装所述部分复值,其中所述部分复值是包含一个或多个常量分量以及一个或多个变量分量的聚合,其中所述第二组指令通过执行从所述常量值高速缓存到运行时存储器中表示所述特定值的分开的位置的所述部分复值的块拷贝以及为了替换所述一个或多个变量分量的、来自所述运行时数据结构的

变量值的一个或多个拷贝来生成所述特定复值。

10.如权利要求1-2中的任何一项权利要求所述的方法，其中将特定值或对所述特定值的引用存储到运行时数据结构上将对所述特定值的所述引用存储到所述运行时数据结构上。

11.如权利要求1-2中的任何一项权利要求所述的方法，其中加载所述常量的所述指令包括一个或多个参量，所述一个或多个参量确定对所述特定值的所述引用是否指向运行时存储器中的可变位置。

12.一个或多个计算机可读媒介，所述一个或多个计算机可读媒介存储指令，所述指令在由一个或多个计算设备执行时，使得权利要求1-11中的任何一项权利要求所述的方法被执行。

13.一种系统，所述系统包括一个或多个计算设备，所述一个或多个计算设备包括至少部分地由计算硬件实现的组件，所述组件被配置为实现权利要求1-11中的任何一项权利要求所述的方法。

使用复常量的方法和系统

[0001] 优先权要求;相关申请

[0002] 本申请是于2014年7月12日提交的、名称为“Complex Constants (复常量)”的美国专利申请No.62/011,411的非临时申请,该申请的全部内容出于所有目的并入本文,如同在此完全阐述一样。

技术领域

[0003] 实施例一般涉及用于在编程语言内支持和/或利用复常量结构体的技术。

背景技术

[0004] 常量表(还被称为常量池)是在程序中存储固定值的数据结构。例如,在Java虚拟机(Java Virtual Machine)的背景下,存储在常量池中的值可以表示在执行的过程中不改变的值,诸如文本(诸如字符串、整数(integer)、双精度浮点数(double)、浮点数(float)等)、字段引用、方法引用、接口引用、名称/类型信息以及其他。常量表的一种用途是向执行程序的运行环境提供与程序员对于常量值的意图有关的信息。

[0005] 然后运行环境可以使用该知识以在静态存储器中创建这些常量值的实例以供程序使用。由于常量值被存储在它们不能被改变的位置中,因此运行环境可以在需要时自由地重复使用这些常量值以避免为了每次使用而重新实例化常量值(或者在一些情况中避免为了垃圾收集而追踪值的需要)。例如,假设程序的源代码定义了两个变量,这两个变量都指代字符串文本“hello world”。在诸如Java虚拟机之类的一些运行环境中,在不使用常量的情况下环境将需要实例化“hello world”两次,并且然后将每个变量赋值到其各自的字符串的存储器位置。然而,如果字符串“hello world”是常量,则操作环境可以自由地作为替代创建“hello world”的一个实例化并且将两个变量都赋值到同一引用。由于“hello world”字符串在存储器中是不可变的,因此不存在操作可能使用第一变量来修改“hello world”字符串并且引发对于第二变量的意外的交叉影响的风险。如果指令试图修改第一变量,则表示修改后的字符串的第二实例化被创建,并且由第一变量存储的引用被转移到修改后的字符串的位置。因此,常量值通过节省存储器以及阻止工作不必要地重复而作为优化程序的执行的重要部分。

[0006] 当前,常量表以及从这些常量表加载常量的过程被定制为适合由运行环境硬编码或本地(native)支持的特定类型的值。这些技术允许运行环境的开发者精调这些常量的处理以及开发特定于特定类型的值的优化。然而,从利用常量值可以获得的运行时效率适用于几乎任何类型的值(诸如复数、元组(tuple)、点(point)、组(group) 等)。将常量值的概念扩张到其他类型的一个问题是对于运行环境(诸如执行程序的虚拟机)的设计者来说预见可能对于软件开发者有用的每种类型的常量值几乎是不可能的。因此,在常量表中表示常量值以及使得这些常量值在运行时存储器中可用的灵活方法将会是在有表现力并且高效执行的程序的追求中的重大进步。

附图说明

[0007] 在附图的图中通过示例的方式而不是通过限制的方式示出了本发明，在附图中相似的附图标记指代相似的元件，并且在附图中：

[0008] 图1是示出了根据各种实施例的在其中可以实现本文所描述的某些技术的示例计算架构的框图。

[0009] 图2是示出了根据实施例的示例类文件的框图。

[0010] 图3是示出了根据实施例的用于虚拟机运行环境的示例架构的框图。

[0011] 图4是示出了根据实施例的用于虚拟机栈上的帧的示例结构的框图。

[0012] 图5是示出了根据实施例的用于常量表的复常量条目的示例结构的框图。

[0013] 图6是示出了根据实施例的常量表201的复常量条目的运行时存储器表示的框图。

[0014] 图7是示出了根据实施例的利用复常量条目创建类文件的示例过程的框图。

[0015] 图8是示出了根据实施例的用于加载复常量的示例过程的框图。

[0016] 图9是示出了适用于实现本文所描述的方法和特征的计算机系统的一个实施例的框图。

具体实施方式

[0017] 在下面的描述中，出于解释的目的，阐述了许多具体细节以便提供对本发明的深入理解。然而，明显可以在没有这些具体细节的情况下实践本发明。在其他实例中，公知的结构和设备以框图形式示出，以便于避免不必要地使本发明模糊。

[0018] 本文根据以下提纲描述实施例：

[0019] 1.0总体概述

[0020] 2.0示例操作环境

[0021] 2.1示例类文件结构

[0022] 2.2示例虚拟机架构

[0023] 2.3加载、链接和初始化

[0024] 3.0复常量

[0025] 3.1类文件表示

[0026] 3.2运行时存储器表示

[0027] 3.3模板化常量

[0028] 3.4示例编译器过程

[0029] 3.5示例虚拟机过程

[0030] 3.6引用vs.值类型

[0031] 3.7常量的可组合性

[0032] 4.0硬件概述

[0033] 5.0扩展和替代物

[0034] 6.0第一附加公开

[0035] 7.0第二附加公开

[0036] 8.0第三附加公开

[0037] 1.0总体概述

[0038] 本文所描述的技术使用来自Java编程语言、Java虚拟机 (“JVM”)和Java运行环境的术语和定义。然而,预期所描述的技术可以结合任何编程语言、虚拟机架构或运行环境来使用。因此,例如诸如“方法”之类的在Java术语中所描述的术语是与诸如“函数”之类的其他术语可互换的。此外,术语“方法”还与术语“类方法”或“对象方法”同义。方法是由名称指代并且可以在使得方法的代码被执行的程序中的各个点处被调用(援用)的一组代码或代码块。术语“援用(invoke)”还与术语“调用(call)”同义。因此,当第一方法“调用”或“援用”第二方法时,这表示第一方法使得第二方法被执行。

[0039] 在实施例中,运行环境提供对于被称为“复常量”的任意类型的常量的支持。因此,运行环境可以甚至对于不是由操作环境“本地”支持的类型提供类似常量的优化和行为。

[0040] 在实施例中,编译器被配置为执行对源代码的分析以及生成较低级的指令,诸如字节码或硬件级指令。在分析期间,编译器确定作为成为常量值的候选者的数据值。例如,用户代码可以包括具体指定值作为常量的注释或语法。作为另一示例,编译器可以检测值的聚合,其中该聚合的至少部分是常量。然后编译器可以指定聚合的常量部分作为成为复常量的候选者。在实施例中,编译器将源代码文件转换为中间表示,诸如字节码。本文呈现的示例假设源语言是诸如Java之类的面向对象的语言,并且因此“类”表示用于一批数据和方法的蓝图 (blueprint),而对象是类的实例化。因此,表示中间表示的文件被称为“类文件”,其中每个单独的文件与特定类相关。然而,本文讨论的技术不限于面向对象的编程语言,而术语“类文件”被用作方便的术语而不是被用作限制。

[0041] 在实施例中,每个类文件由诸如虚拟机之类的运行环境处理,以在运行时存储器中生成相关联的类的表示。例如,运行环境可以被配置为在类由程序第一次使用时加载该类。响应于该使用,运行环境解析相关联的类文件、为常量、静态字段、类的方法代码等分配存储器,然后初始化这些值。在条目来自常量表的情况中,为这些常量分配的存储器被保护以使得这些存储器区域不能被修改。下文在节2.3“加载、链接和初始化”中提供了更深入的解释。

[0042] 在实施例中,类文件包括识别程序内的各种常量值的常量表。例如,常量表中的每个条目可以表示固定的整数、双精度浮点数、浮点数、字符串、长整数、方法引用、字段引用、接口引用等等。在实施例中,表示复常量的条目包括根据其构造常量的原始数据以及对汇编来自原始数据的复常量的汇编器方法的引用。取决于实施例,原始数据可以被直接存储在常量表条目中,或者原始数据可以被存储在其他常量表条目中并且用对常量表的引用或索引来表示。在一些实施例中,常量表具有被称为“字节型”的条目类型,该条目类型表示用于存储数据的原始字节的容器。取决于实施例,汇编器方法可以从源代码导出、由编译器自动供应或者由运行环境维护。然而,出于安全的目的,一些实施例可以选择使汇编器方法仅从选择数量的源供应或者甚至仅从运行环境本身供应。在后种情况中,运行环境的开发者将仍然需要供应对于各种类型的复常量的支持。然而,由于用于任意类型的常量的常量表条目已经是可用的,因此复常量提供在不需要创建新条目类型的情况下扩展所支持的常量的方便的框架。

[0043] 在实施例中,当复常量首次由指令引用时,运行环境执行汇编器方法。汇编器方法读取原始数据并且生成复常量的值。然后该值被存储在复值高速缓存中。因此,对复常量的将来引用可以通过返回被高速缓存的值而不是重复执行汇编器方法以重新构造该值的工

作来提供 (serve)。

[0044] 在一些实施例中,由复常量条目引用的汇编器方法是元工厂 (metafactory)方法,当该元工厂方法被执行时创建返回复常量的值的工厂方法。实施例可以选择实现元工厂技术的一个原因是为了提供对于模板化常量的支持。在一些情况中,源代码可以包括重复使用聚合的大多数分量部分的操作或方法,但是其中有一些相对较小的改变。例如,源代码可以包括采用变量x作为输入并且构造数组[1,2,x,4]的方法。在不使用模板化常量技术的情况下,当接收到用于x的值“3”时,运行环境通过在堆上分配用于四个元素的空间并且然后将每个值插入到其各自的索引中以创建数组[1,2,3,4]来构造前面提到的数组。在诸如 Java之类的一些环境中,数组被表示为Object(对象),并且因此分配还可以包括用于除了数据元素之外的头部信息的空间。在该示例中,运行环境在常量表中为整数存储条目,运行环境使用这些条目来将“1”、“2”和“4”单独复制到数组中的它们各自的位置。然而,如果利用用于x的不同的值来构造另一数组,则将重复相同的过程以创建新数组。

[0045] 在上面的示例中,数组的第一、第二和第四索引是常量并且在数组实例化之间被重复使用。通过利用模板化常量,运行环境可以重复使用在创建第一数组时完成的工作中的一些工作以创建随后的数组。这通过定义元工厂方法来实现,该元工厂方法采用复值数据作为输入、构造并且高速缓存具有常量和占位符值的数组,然后生成将高速缓存的数组与供应的变量值组合的工厂方法。因此,对于上面的示例,编译器创建指定整数[1]、[2]和[4]的常量复条目。对汇编器方法的引用是采用前面提到的数据并且构造数组[1,2,x,4]的元工厂,其中x是通过将值从操作数栈出栈而供应的值。然后该部分数组作为被高速缓存的复值(complex value)而被存储。附加地,元工厂方法生成并且执行被称为工厂方法的字节码指令,该工厂方法执行将被高速缓存的复值复制到新数组中的块拷贝,然后通过将变量值出栈而在合适的位置中执行这些值的单独拷贝。然后通过将对数组的引用推到操作数栈上来返回新数组。一旦生成工厂,则对作为用于复常量的汇编器方法的元工厂的引用被替换为对工厂的引用。因此,加载常量结果的将来请求由工厂提供,而不是通过重新执行元工厂来提供。

[0046] 由于块拷贝一般是比多个单独拷贝更廉价的操作,因此运行环境可以高效地重复使用在创建第一数组时完成的工作中的至少部分工作以创建将来的数组。例如,如果数组由10000个元素组成,这些元素中仅两个元素由变量供应,则运行环境可以用被高速缓存的值的块拷贝指令以及用所供应的变量的值来替换占位符的两个单独的拷贝指令来替换10000个单独的拷贝指令,因此,可以获得运行时效率的显著提高。在数组方面描述了前面提到的示例。然而,模板化常量技术可以被应用于诸如数组、组、元组、通用对象(general object)等等之类的任何种类的聚合而没有限制。在一些实施例中,为了支持设计的一致性,甚至针对不具有变量分量的常量使用上文所描述的元工厂/ 工厂技术。在这样的情况中,工厂方法仅返回被高速缓存的值,而不将该被高速缓存的值与来自操作数栈的变量值合并。

[0047] 在一些情况中,使用复常量降低客户代码的整体大小。例如,Java 中的程序可以具有初始化数组[1,2,3]的类,该类将使得用于“创建三个元素的数组、然后将1存储到第一元素、然后将2存储到第二元素、然后将3存储到第三元素”的字节码指令被生成。这比依赖于编译器创建用于常量数组的复常量条目以及输出加载由该条目表示的常量的单个字节

码指令冗长得多。

[0048] 2.0示例操作架构

[0049] 图1示出了在其中可以实践本文所描述的技术的示例计算架构 100。

[0050] 如图1所示，计算架构100包括源代码文件101，源代码文件101 由编译器102编译为表示要被执行的程序的类文件103。然后执行平台112加载并且执行类文件103，执行平台112包括运行环境113、操作系统111以及实现运行环境113和操作系统111之间的通信的一个或多个应用编程接口(API)110。运行环境113包括虚拟机104，虚拟机104包括各种组件，诸如存储器管理器105(其可以包括垃圾收集器)、检查类文件103和方法指令的有效性的验证器106、定位和建立类的存储器中(in-memory)表示的类加载器107、用于执行虚拟机104代码的解释器108、用于产生优化后的机器级别代码的即时 (JIT)编译器109以及用于将符号引用解析为类和/或方法的链接解析器114。

[0051] 在实施例中，计算架构100包括源代码文件101，源代码文件101 包含以诸如Java、C、C++、C#、Ruby、Perl等之类的特定编程语言编写的代码。因此，源代码文件101遵守用于相关联的语言的语法和/ 或语义规则的特定集合。例如，以Java编写的代码遵守Java语言规范。然而，由于规范随时间更新和修订，因此源代码文件101可以与版本号相关联，该版本号指示源代码文件101所遵守的规范的修订。用于编写源代码文件101的确切的编程语言一般不关键。

[0052] 在各种实施例中，编译器102将根据依照程序员的方便的规范编写的源代码转换成由特定机器环境直接可执行的机器代码或对象代码，或者转换成由能够在各种特定机器环境上运行的虚拟机104可执行的中间表示(“虚拟机代码/指令”)，诸如字节码。虚拟机指令可由虚拟机104以比源代码更直接和高效的方式执行。将源代码转换为虚拟机指令包括将来自语言的源代码功能映射到利用诸如数据结构之类的底层资源的虚拟机功能。通常，程序员经由源代码以简单术语呈现的功能被转换成更复杂的步骤，该更复杂的步骤更直接地映射到由虚拟机104驻留在其上的底层硬件所支持的指令集。

[0053] 一般来说，程序作为编译后的程序或者解释后的程序执行。当程序被编译时，在执行之前代码从第一语言被全局地转变为第二语言。由于转变代码的工作是提前执行的，因此编译后的代码往往具有杰出的运行时性能。附加地，由于转变在执行之前全局地发生，因此可以使用诸如常量折叠、死代码消除、内联等之类的技术来分析和优化代码。然而，取决于被执行的程序，启动时间可以是显著的。附加地，插入新代码将要求程序离线、重新编译和重新执行。当程序被解释时，在程序执行的同时程序的代码被逐行读取并且转换为机器级别的指令。因此，程序具有短的启动时间(可以几乎立刻开始执行)，但是通过即时(on the fly)执行转换减弱了运行时性能。此外，由于每个指令被单独分析，因此依赖于程序的更全局的分析的许多优化不能被执行。

[0054] 在一些实施例中，虚拟机104包括解释器108和JIT编译器109 (或者实现二者的方面的组件)，并且使用解释和编译技术的组合来执行程序。例如，虚拟机104可以最初开始于经由解释器108解释表示程序的虚拟机指令同时追踪关于程序行为的统计数据，诸如虚拟机 104执行不同的代码部分或代码块的频率。一旦代码块超过某一阈值 (是“热的”)，则虚拟机104调用JIT编译器109来执行块的分析并且生成替换“热”代码块的优化的机器级别的指令以供将来执行。由于程序往往花费它们的时间的大部分来执行它们的整体代码的小部

分，因此仅编译程序的“热”部分可以提供与完全编译的代码相似的性能，但是没有启动惩罚(start-up penalty)。

[0055] 为了提供清楚的示例，源代码文件101被示为要由执行平台111 执行的程序的“最高级别”表示。然而，尽管计算架构100将源代码文件101描绘为“最高级别”程序表示，但是在其他实施例中源代码文件101可以是经由“较高级别”编译器接收的中间表示，该“较高级别”编译器将不同语言的代码文件处理成源代码文件101的语言。为了示出清楚的示例，以下公开假设源代码文件101遵守基于类的面向对象的编程语言。然而，这不是利用本文所描述的特征的要求。

[0056] 在实施例中，编译器102接收源代码文件101作为输入并且将源代码文件101转换成以虚拟机104所期望的格式的类文件103。例如，在JVM的背景下，Java虚拟机规范第4章定义了期望类文件103遵守的特定类文件格式。在一些实施例中，类文件103包含已经从源代码文件101转换而来的虚拟机指令。然而，在其他实施例中，类文件 103还可以包含其他结构，诸如识别关于各种结构(类、字段、方法等)的元数据和/或常量值的表。

[0057] 以下讨论将假设类文件103中的每个类文件表示源代码文件101 中定义的(或者由编译器102或虚拟机104动态生成的)各自的“类”。然而，前面提到的假设不是严格的要求并且将取决于虚拟机104的实现。因此，可以不管类文件103的确切格式而仍然执行在此所描述的技术。在一些实施例中，类文件103被划分成一个或多个“库”或“包 (package)”，该一个或多个“库”或“包”中的每个包括提供相关功能的类的汇集。例如，库可以包含实现输入/输出(I/O)操作、数学工具、密码技术、图形实用程序等的一个或多个类文件。此外，一些类(或者这些类内的字段/方法)可以包括将它们的使用限制在特定的类/库/包内或者限制到具有合适的许可的类的访问限制。

[0058] 2.1示例类文件结构

[0059] 图2示出了根据实施例的以框图形式的用于类文件200的示例结构。为了提供清楚的示例，本公开的剩余部分假设计算架构100的类文件103遵守本节中所描述的示例类文件200的结构。然而，在实践环境中，类文件200的结构将取决于虚拟机104的实现。此外，在此讨论的一个或多个特征可以修改类文件200的结构以例如添加附加结构类型。因此，类文件200的确切结构对于本文所描述的技术来说不是关键的。就节2.1而言，“类”或“该类”指代由类文件200表示的类。

[0060] 在图2中，类文件200包括常量表201、字段结构208、类元数据204和方法结构209。

[0061] 在实施例中，常量表201是充当用于类的符号表以及实现其他功能的数据结构。例如，常量表201可以存储与源代码文件101中使用的各种识别符(诸如类型、范围、内容和/或位置)相关的数据。常量表201具有用于由编译器102从源代码文件101导出的值结构202(表示整型(int)、长整型、双精度浮点型(double)、浮点型、字节型、字符串型等类型的常量值)、类信息结构203、名称和类型信息结构 205、字段引用结构206以及方法引用结构207的条目。在实施例中，常量表201被实现为将索引i映射到结构j的数组。然而，常量表201 的确切实现不是关键的。

[0062] 在一些实施例中，常量表201的条目包括索引其他常量表201条目的结构。例如，用于表示字符串的值结构202中的一个值结构的条目可以保持将其“类型”识别为字符串的标签以及对存储表示该字符串的ASCII字符的字符型、字节型或整型值的常量表201的一个或

多个其他值结构202的索引。

[0063] 在实施例中，常量表201的字段引用结构206保持对常量表201 中的表示定义字段的类的类信息结构203中的一个类信息结构的索引以及对常量表201中提供字段的名称和描述符的名称和类型信息结构 205中的一个名称和类型信息结构的索引。常量表201的方法引用结构207保持对常量表201中的表示定义方法的类的类信息结构203中的一个类信息结构的索引以及对常量表201中提供用于方法的名称和描述符的名称和类型信息结构205中的一个名称和类型信息结构的索引。类信息结构203保持对常量表201中的保持相关联的类的名称的值结构202中的一个值结构的索引。名称和类型信息结构205保持对常量表201中的存储字段/方法的名称的值结构202中的一个值结构的索引以及对常量表201中的存储描述符的值结构202中的一个值结构的索引。

[0064] 在实施例中，类元数据204包括用于类的元数据，诸如版本号(多个版本号)、常量池中的条目的数量、字段的数量、方法的数量、访问标志(类是否是公有的(public)、私有的(private)、最终的(final)、抽象的(abstract)等)、对识别该类的常量表201的类信息结构203 中的一个类信息结构的索引、对识别超类(如果有的话)的常量表201 的类信息结构203中的一个类信息结构的索引等等。

[0065] 在实施例中，字段结构208表示识别类的各个字段的一组结构。字段结构208为类的每个字段存储用于该字段的访问器标志(字段是否是静态的(static)、公有的(public)、私有的(private)、最终的(final)等)、对常量表201中的保持字段的名称的值结构202中的一个值结构的索引、以及对常量表201中的保持字段的描述符的值结构202中的一个值结构的索引。

[0066] 在实施例中，方法结构209表示识别类的各种方法的一组结构。方法结构209为类的每种方法存储用于该方法的访问器标志(方法是否是静态的(static)、公有的(public)、私有的(private)、同步的(synchronized)等)、对常量表201中的保持方法的名称的值结构202中的一个值结构的索引、对常量表201中的保持方法的描述符的值结构202中的一个值结构的索引、以及对应于如源代码文件101 中定义的方法的主体的虚拟机指令。

[0067] 在实施例中，描述符表示字段或方法的类型。例如，描述符可以被实现为遵守特定语法的字符串。尽管确切的语法不是关键的，但是下文描述了几个示例。

[0068] 在描述符表示字段的类型的示例中，描述符识别由字段保持的数据的类型。在实施例中，字段可以保持基本类型、对象或数组。当字段保持基本类型时，描述符是识别该基本类型的字符串(例如，“B”= byte(字节型)、“C”=char(字符型)、“D”=double(双精度浮点型)、“F”=float(浮点型)、“I”=int(整型)、“J”=long int(长整型)等)。当字段保持对象时，描述符是识别对象的类名称的字符串(例如，“L ClassName”)。在该情况中“L”指示引用，因此“L ClassName”表示对类ClassName的对象的引用。当字段是数组时，描述符识别由数组保持的类型。例如，“[B”指示字节型的数组，其中“[”指示数组而“B”指示该数组保持字节型的基本类型。然而，由于数组可以被嵌套，因此用于数组的描述符还可以指示嵌套。例如，“[[L ClassName”指示数组，在该数组中每个索引保持保持类 ClassName的对象的数组。在一些实施例中，ClassName是完全限定的并且包括类的简单名称以及类的路径名称。例如，ClassName可以指示文件被存储在托管类文件200的包、库或文件系统中的何处。

[0069] 在方法的情况中，描述符识别方法的参量(parameter)和方法的返回类型。例如，

方法描述符可以遵循一般形式“({ParameterDescriptor})ReturnDescriptor”，其中{ParameterDescriptor}是表示参量的字段描述符的列表，而 ReturnDescriptor是识别返回类型的字段描述符。例如，字符串“V”可以被用于表示空(void)返回类型。因此，在源代码文件101中定义为“Object m(int I,double d,Thread t){…}”的方法匹配描述符“(I D L Thread)L Object”。

[0070] 在实施例中，方法结构209中保持的虚拟机指令包括引用常量表 201的条目的操作。

[0071] 使用Java作为示例，考虑以下类：

[0072]
```
class A
{
int add12and13() {
    return B.addTwo(12, 13);
  }
}
```

[0073] 在上面的示例中，Java方法add12and13在类A中定义、不带参数并且返回整数。方法add12and13的主体调用采用常量整数值12和13 作为参数的类B的静态方法addTwo，并且返回结果。因此，在常量表201中，编译器102包括对应于对方法B.addTwo的调用的方法引用结构以及其他条目。在Java中，对方法的调用(call)向下编译为 JVM的字节码中的调用(invoke)命令(在该情况中，该调用命令是 invokestatic，因为addTwo是类B的静态方法)。调用命令被提供了对应于识别定义addTwo的类“B”、addTwo的名称“addTwo”以及 addTwo的描述符“(I I)I”的方法引用结构的对常量表201的索引。例如，假设前面提到的方法引用被存储在索引4处，那么字节码指令可以表现为“invokestatic#4”。

[0074] 由于常量表201利用承载识别信息的结构以符号形式指代类、方法和字段，而不是利用对存储器位置的直接引用指代类、方法和字段，因此常量表201的条目被称为“符号引用”。符号引用被用于类文件 103的一个原因是因为在一些实施例中编译器102不了解一旦类被加载到运行环境113中，这些类将怎样被存储以及将被存储在何处。如将在节2.3中描述的，在被引用的类(以及相关联的结构)已经被加载到运行环境中并且已经被分配了具体存储器位置之后，最终符号引用的运行时表示由虚拟机104解析为具体的存储器地址。

[0075] 2.2示例虚拟机架构

[0076] 图3示出了根据实施例的以框图形式的示例虚拟机存储器布局 300。为了提供清楚的示例，剩余的讨论将假设虚拟机104遵守图3 中描绘的虚拟机存储器布局300。附加地，尽管虚拟机存储器布局300 的组件可以被称为存储器“区域”，但是不要求存储器区域是相邻的。

[0077] 在图3示出的示例中，虚拟机存储器布局300被划分成共享区域 301和线程区域307。

[0078] 共享区域301表示存储在虚拟机104上执行的各种线程之间共享的结构的存储器中的区域。共享区域301包括堆302和每类(per-class) 区域303。在实施例中，堆302表示从

中分配用于类实例和数组的存储器的运行时数据区域。在实施例中，每类区域303表示存储属于各个类的数据的存储器区域。在实施例中，对于每个加载的类，每类区域303包括表示来自类的常量表201的数据的运行时常量池304、字段和方法数据306(例如，以保持类的静态字段)以及表示用于类的方法的虚拟机指令的方法代码305。

[0079] 线程区域307表示存储特定于各个线程的结构的存储器区域。在图3中，线程区域307包括表示由不同线程利用的每线程结构的线程结构308和线程结构311。为了提供清楚的示例，图3中描绘的线程区域307假设两个线程正在虚拟机104上执行。然而，在实践环境中，虚拟机104可以执行任何任意数量的线程，其中线程结构的数量相应地缩放。

[0080] 在实施例中，线程结构308包括程序计数器309和虚拟机栈310。类似地，线程结构311包括程序计数器312和虚拟机栈313。在实施例中，程序计数器309和程序计数器312存储正在由其各自的线程执行的虚拟机指令的当前地址。因此，当线程逐句通过(step through) 指令时，程序计数器被更新以维护对当前指令的索引。在实施例中，虚拟机栈310和虚拟机栈313各自存储用于保持局部变量和部分结果的其各自的线程的帧，并且还被用于方法调用和返回。

[0081] 在实施例中，帧是用于存储数据和部分结果、返回用于方法的值以及执行动态链接的数据结构。每次调用方法时创建新帧。当使得帧被生成的方法完成时，帧被销毁。因此，当线程执行方法调用时，虚拟机104生成新帧，并且将该帧推到与线程相关联的虚拟机栈上。当方法调用完成时，虚拟机104将方法调用的结果传递回前一帧并且使当前帧出栈。在实施例中，对于给定的线程，在任何时间点处一个帧是活动的。该活动的帧被称为当前帧，引发当前帧的生成的方法被称为当前方法，而当前方法所属的类被称为当前类。

[0082] 图4示出了根据实施例的以框图形式的示例帧400。为了提供清楚的示例，剩余的讨论将假设虚拟机栈310和虚拟机栈313的帧遵守帧400的结构。

[0083] 在实施例中，帧400包括局部变量401、操作数栈402和运行时常量池引用表403。

[0084] 在实施例中，局部变量401被表示为各自保持诸如布尔型、字节型、字符型、短整型、整型、浮点型、引用型等之类的值的变量的数组。此外，诸如长整型或双精度浮点型之类的一些值类型可以由数组中的多于一个条目表示。局部变量401被用于在方法调用时传递参量以及存储部分结果。例如，当响应于调用方法而生成帧400时，参量可以被存储在局部变量401内的预定义的位置中，诸如对应于调用中的第一个到第N个参量的索引1-N。

[0085] 在实施例中，当虚拟机104创建帧400时，操作数栈402默认是空的。然后虚拟机104供应来自当前方法的方法代码305的指令以将来自局部变量501的常量或值加载到操作数栈502上。其他指令采用来自操作数栈402的操作数、对它们进行操作并且将结果推回到操作数栈402上。此外，操作数栈402被用于准备要被传递到方法的参量以及接收方法结果。例如，在发出对方法的调用之前，被调用的方法的参量可以被推到操作数栈402上。然后虚拟机104生成用于方法调用的新帧，其中前一帧的操作数栈402上的操作数出栈并且被加载到新帧的局部变量401中。当被调用的方法终止时，新帧从虚拟机栈出栈并且返回值被推到前一帧的操作数栈402上。

[0086] 尽管使用诸如“数组”和/或“栈”之类的数据结构来指代局部变量401和操作数栈402，但是对用于实现这些元素的数据结构的类型没有限制。附加地，参照局部变量401和操作数栈402在此指代的数据结构涉及数据结构的高级别表示。实施例可以使用各种较低级

别的存储机制来实现这些数据结构，诸如将局部变量401和/或操作数栈 402的一个或多个值存储在执行虚拟机104的机器硬件的中央处理单元(CPU)的一个或多个寄存器中。

[0087] 在实施例中，运行时常量池引用表403包含对当前类的运行时常量池304的引用。运行时常量池引用表403被用于支持解析。解析是由此运行时常量池304中的符号引用被翻译为具体存储器地址的过程，该过程按需加载类以解析尚未定义的符号并且将变量访问翻译成与这些变量的运行时位置相关联的存储结构中的合适的偏移。

[0088] 2.3加载、链接和初始化

[0089] 在实施例中，虚拟机104动态加载、链接和初始化类。加载是寻找具有特定名称的类并且在运行环境113的存储器内创建来自该类的相关联的类文件200的表示的过程。例如，在虚拟机存储器布局300 的每类区域303内为该类创建运行时常量池304、方法代码305以及字段和方法数据306。链接是采用类的存储器中表示并且将其与虚拟机104的运行时状态组合以使得类的方法可以被执行的过程。初始化是执行类构造器以设置类的字段和方法数据306的起始状态和/或为了被初始化的类在堆302上创建类实例的过程。

[0090] 以下是可以由虚拟机104实现的加载、链接和初始化技术的示例。然而，在许多实施例中步骤可以交错，以使得初始类被加载，然后在链接期间第二个类被加载以解析在第一个类中发现的符号引用，这又使得第三个类被加载，等等。因此，通过加载、链接和初始化的阶段的进程可以依据类而不同。此外，一些实施例可以延迟(“懒惰地”执行)加载、链接和初始化过程中的一个或多个功能直至实际需要该类。例如，方法引用的解析可以被延迟直至调用被引用的方法的虚拟机指令被执行。因此，对于每个类步骤何时执行的确切时机在实施方式之间可以差别很大。

[0091] 为了开始加载过程，虚拟机104通过调用加载初始类的类加载器 107而启动。指定初始类的技术将依据实施例变化。例如，一种技术可以使虚拟机104在启动时接受指定初始类的命令行参数。

[0092] 为了加载类，类加载器107解析对应于类的类文件200并且确定类文件200是否为形式良好的(满足虚拟机104的语法期待)。如果类文件200不是形式良好的，则类加载器107生成错误。例如，在Java 中，错误可以以异常的形式生成，该异常被抛出到异常处理器以供处理。否则，类加载器107通过在每类区域303内分配用于该类的运行时常量池304、方法代码305以及字段和方法数据306来生成该类的存储器中表示。

[0093] 在一些实施例中，当类加载器107加载类时，类加载器107还递归加载被加载的类的超类。例如，虚拟机104可以确保在继续用于特定类的加载、链接和初始化过程之前，该特定类的超类被加载、链接和/或初始化。

[0094] 在链接期间，虚拟机104验证类、准备类并且执行类的运行时常量池304中定义的符号引用的解析。

[0095] 为了验证类，虚拟机104检查类的存储器中表示在结构上是否是正确的。例如，虚拟机104可以检查除了泛型类对象(Object)之外的每个类具有超类，检查最终类不具有子类以及最终方法没有被重写，检查常量池条目是否彼此一致，检查当前类是否具有用于运行时常量池304中所引用的类/字段/结构的正确的访问许可，检查方法的虚拟机104代码将不会引发意外行为(例如，确保跳转指令不会使得虚拟机104超出方法的末尾)，等等。在验证期间执行的确切检查取决于虚拟机104的实现。在一些情况中，验证可以引发附加的类被

加载，但是在继续之前不一定要求这些类还被链接。例如，假设类A包含对类B的静态字段的引用。在验证期间，虚拟机104可以检查类B以确保所引用的静态字段实际存在，这可以引发类B的加载，但不一定引发类B的链接或初始化。然而，在一些实施例中，某些验证检查可以被推迟至较晚的阶段，诸如在符号引用的解析期间被检查。例如，一些实施例可以推迟检查用于符号引用的访问许可直至这些引用被解析。

[0096] 为了准备类，虚拟机104将位于用于类的字段和方法数据306内的静态字段初始化为默认值。在一些情况中，将静态字段设置为默认值可以与运行用于该类的构造器不同。例如，验证过程可以在初始化期间将静态字段清零或者将静态字段设置为构造器将期待这些字段具有的值。

[0097] 在解析期间，虚拟机104从包括在类的运行时常量池304中的符号引用动态地确定具体存储器地址。为了解析符号引用，虚拟机104 利用类加载器107来加载符号引用中识别的类(如果还未加载)。一旦该类被加载，则虚拟机104了解所引用的类及其字段/方法的每类区域303内的存储器位置。然后虚拟机104将控制传递到链接解析器 114，链接解析器114将符号引用替换为对所引用的类、字段或方法的具体存储器位置的引用。例如，链接解析器114可以查询元数据、表或其他信息以搜索和定位具体存储器位置。在实施例中，链接解析器 114高速缓存解析以便一旦在程序的执行期间再次遇到相同的类/名称 /描述符则重复使用该解析。在一些实施例中，通过替换类的运行时常量池304内的符号引用来执行高速缓存。然而，在其他实施例中，分开的高速缓存数据结构被用于存储指向具体存储器位置的指针。

[0098] 在一些实施例中，在链接期间解析符号引用的步骤是可选的。例如，实施例可以以“懒惰的”方式执行符号解析，从而延迟解析的步骤直至需要所引用的类/方法/字段的虚拟机指令被执行。

[0099] 在初始化期间，虚拟机104执行类的构造器以设置该类的起始状态。例如，初始化可以初始化用于类的字段和方法数据306以及在由构造器创建的堆302上生成/初始化任何类实例。例如，用于类的类文件200可以指定特定方法是用于设立起始状态的构造器。因此，在初始化期间，虚拟机104执行该构造器的指令。

[0100] 在一些实施例中，虚拟机104通过初始地检查字段/方法是否在所引用的类中定义来执行对字段和方法引用的解析。否则，虚拟机104 针对所引用的字段/方法递归搜索所引用的类的超类直至字段/方法被定位或者到达最高级别的超类，在到达最高级别的超类的情况中生成错误。

[0101] 3.0复常量

[0102] 在实施例中，运行环境提供对于被称为“复常量”的任意类型的常量的支持。因此，运行环境可以甚至为不是“本地”支持的类型提供类似常量的优化和行为。

[0103] 例如，考虑映射/散列/字典常量的情况。在诸如Java之类的许多编程语言中，将包含键值对的Map<String,String>传递到应用编程接口(API)的初始化是常见的。源代码文件101中的代码可以看上去为：

[0104] Map<String,String>config＝new HashMap<>();

[0105] config.put(“case.sensitive”,“false”);

[0106] config.put(“file.root”,“/path/to/file/root”);

[0107] FileReader reader＝FileReader.make(config)；

[0108] 当由编译器102编译时，上面的示例代码变成许多虚拟机指令以创建Map(映射)、填充Map以及将Map抛弃(例如，显式地释放 Map的存储器或者经由垃圾收集器清理)。通过使用“复常量”，实施例可以将映射编码为字符串(常量)和从字符串生成“Map常量”的聚合方法。

[0109] 因此，这样的实施例中的源代码101可以允许程序员用缩减数量的指令来表示“Map常量”，诸如：

[0110] Map<String,String>config＝#{“case.sensitive”＝>“false”, “file.root”＝>“/path/to/file/root”}；

[0111] 并且编译器102可以将该语句翻译成复常量。由于复常量由运行环境113的常量表/池追踪并且被不可变地存储在存储器中，因此虚拟机104在需要时可以自由地重复使用Map常量，而不必重做重新创建 /释放映射常量的工作。因此，将这样的结构表示为复常量减少了代码的大小以及允许运行效率的提升。

[0112] 尽管上面的示例使用了映射常量，但是复常量可以被应用于任何数量的常量结构而没有限制，诸如复数、元组、数组、列表、组、序列、泛型对象等。

[0113] 3.1类文件表示

[0114] 在实施例中，复常量以可一般化为几乎任何种类的常量值的格式在类文件200的常量表201中表示。

[0115] 图5是示出了根据实施例的示例复值条目500的框图。在图5中，复值条目500是常量表201的值结构202的成员。复值条目500包括复值数据501和复值汇编器指令502。在图5中，仅显示了一个复值条目500以避免不必要地使示意图模糊。然而，类文件200可以包含几乎任何数量的复值条目。

[0116] 在实施例中，复值数据501表示复值汇编器指令502从其构造复常量的值或部分值的数据。在一些实施例中，复值数据501存储复常量的原始数据。例如，如果复常量是数组，则复值数据501将存储组成数组的值。然而，在其他实施例中，复值数据501替代地存储对常量表201的其他条目的引用或索引。例如，如果构造数组[1,2,3,4]，则复值条目500可以存储对常量表201中的表示单个整数[1]、[2]、[3] 和[4]的其他值结构202的索引。在一些实施例中，常量表201的值结构202包括用于将字节型存储为用于原始数据的容器的条目类型。因此，在这样的实施例中，复值数据501包括对保持用于复值条目500 的原始数据的字节条目的引用。在一些实施例中，编译器102使用各种技术来存储复值数据501并且基于诸如原始数据的大小之类的复常量的特性来选择使用哪种技术。下文在节3.4“示例编译器过程”中描述了关于构造复值条目500的编译器102的过程的附加的示例。

[0117] 在实施例中，复值汇编器指令502表示元工厂方法，该元工厂方法读取复值数据501、将复值数据501处理成复常量值、在复值高速缓存中高速缓存复常量值以及生成返回被高速缓存的值的工厂方法。在一些实施例中，复值高速缓存是在复值汇编器指令502的执行时用于存储被高速缓存的复值(或者对其的引用)的、由虚拟机104预留的存储器的区域。在一些情况中，诸如对于下文在节3.3“模板化常量”中以更多细节描述的模板化常量来说，工厂方法还在返回组合的值之前将被高速缓存的值与操作数栈402上的附加变量值组合。在一些实施例中，复值汇编器指令502存储对存储指令或保持对要由运行环境 113解析的

指令的符号引用的方法引用结构207的条目的常量表的另一条目的索引，而不是存储指令本身。

[0118] 在一些实施例中，在输入是常量(或者“模板化常量”的情况中的半常量)并且聚合方法为确定性(对于相同输入重复执行汇编器方法产生“等价的”输出)的情况中，“复常量”是有用的。例如，如果汇编器方法是非确定性的，诸如汇编器方法利用随机数生成器来使用输入，则非确定性汇编器的返回值不能被有用地高速缓存。然而，用于“复常量”的框架仍然可以是用于汇编器方法为非确定性的情况的有用的组织框架。

[0119] 取决于实施例，复值汇编器指令502可以由源代码文件101供应、由编译器102在编译源代码文件101时自动生成或者由虚拟机104提供。在从运行环境113之外的源提供复值汇编器指令502的情况中，这允许源代码文件101的编写者或者编译器102的开发者提供用于常量类型的他们自己的行为和定义。然而，出于安全目的，一些实施例可以将复值汇编器指令502的源限制为可信源。因此，运行环境113 可以接受由编译器102生成的复值汇编器指令502，但是可以保持生成常量的过程对编写源代码的用户不可见。在一些情况中，虚拟机104 可以仅接受在由运行环境113提供的库中的复值汇编器指令502。在这样的情况中，复常量的范围将被限制为由运行环境113的开发者设想的那些复常量，但是本文所描述的技术对于提供可以通过向库添加附加的元工厂方法来方便地扩展到其他常量类型的框架来说仍然是有用的。因此，不存在创建用于随后支持的每个类型的常量的不同的常量表201条目类型的要求。

[0120] 3.2运行时存储器表示

[0121] 上文在节2.3“加载、链接和初始化”中描述了用于将类(包括常量表201)加载到虚拟机存储器布局300中的一般过程的示例。本节关注虚拟机104的虚拟机存储器布局300中的复常量的表示。

[0122] 图6是示出了根据实施例的常量表201的复值条目500的运行时存储器表示的框图。在图6中，运行时复常量条目600包括运行时复值数据601、复值汇编器引用602和复值高速缓存引用603。

[0123] 在实施例中，运行时复值数据601是由虚拟机104已经加载到虚拟机存储器布局300中的类文件200的复值数据501。取决于实施例，运行时复值数据601可以存储用于复常量的实际数据或对来自常量表 201的条目的其他运行时表示的引用。

[0124] 在实施例中，复值汇编器引用602初始是对存储在方法代码305 中的元工厂方法604的引用。元工厂方法604表示已经加载到运行时存储器中的复值汇编器指令502。在一些实施例中，对元工厂方法604 的复值汇编器引用602是在复常量的第一次使用时由虚拟机104解析的符号引用。尽管元工厂方法604被描绘为利用复常量的同一类的方法代码305的部分，但是其他实施例可以在其他类或者在由运行环境供应的特殊库中存储元工厂方法。在实施例中，复值汇编器引用602 初始引用元工厂方法604。然而，在复常量的第一次使用时，虚拟机 104执行元工厂方法604以填充复值高速缓存引用603并且生成工厂方法605。然后虚拟机104将复值汇编器引用602更新为工厂方法605 以用于复常量的将来加载。尽管工厂方法605被示为被放置为方法代码305的部分，但是该位置不是关键的。例如，一些实施例可以在堆 302或虚拟机存储器布局300中的任何其他区域上存储工厂方法605。在一些实施例中，元工厂方法604供应代码以填充复值高速缓存引用 603、更新复值汇编器引用602以

指向所生成的工厂方法605以及执行工厂方法605。然而，在其他实施例中，前面提到的步骤中的一个或多个步骤由与元工厂方法605的代码分开的、属于虚拟机104的代码执行。

[0125] 在实施例中，复值高速缓存引用603是对通过执行由复值汇编器引用602初始引用的604而创建的用于复常量的被高速缓存的值的引用。因此，在一些实施例中，复值高速缓存引用603初始为空并且被更新为指向由元工厂方法305生成的值的位置。在图6的示例中，复值高速缓存引用603引用被分配和存储在堆302上的被高速缓存的值。对于诸如Java之类的一些语言来说，诸如数组、对象等之类的聚合是引用类型并且在堆302上实例化。因此，图6是元工厂方法604在堆 302上实例化被高速缓存的复值并且在复值高速缓存引用603中存储对该位置的引用的情况的示例。然而，在其他实施例中，被高速缓存的值可以在运行时常量复条目600中存储在位，而不是通过引用来存储。此外，取决于被表示的复常量的大小和复杂度，一些实施例在这两种表示之间动态切换。

[0126] 3.3模板化常量

[0127] 在实施例中，运行环境113支持用于如下的值的创建的“模板化常量”：该值的部分是常量，其余部分由一个或多个变量供应。

[0128] 在实施例中，通过将这样的部分常量转换成复值条目来支持模板化常量，在该复值条目中复值数据501识别聚合的常量和非常量部分，而复值汇编器指令502被配置为高速缓存具有占位符值的中间结果。在这样的实施例中，通过执行由复值汇编器指令502表示的元工厂方法604而生成的工厂方法605被配置为执行被高速缓存的值的块拷贝，然后执行附加的单独拷贝以用由变量供应的值来填充占位符值。例如，通过将值从操作数栈402出栈。

[0129] 考虑以下示例代码：

[0130] int x=…

[0131] int[]a={1,4,x,10};

[0132] 在上面的示例代码中，a是四个元素的数组，其中第一、第二和第四索引是常量，而第三索引由变量x供应。

[0133] 在实施例中，编译器102在复值条目500中表示上面的数组，其中复值数据501指定第一索引是整数“1”、第二索引是整数“4”、第三索引将由变量供应以及第四索引是整数“10”。由复值汇编器指令502表示的元工厂方法604被配置为读取复值数据501以及实例化/高速缓存{1,4,P,10}的数组，其中P是为占位符预留和/或与指示它是占位符的元数据相关联的值。附加地，复值汇编器指令502被配置为生成工厂方法605，工厂方法605通过执行从被高速缓存的数组到新数组的块拷贝以及执行从变量x到第三索引中的单独拷贝来实例化新数组。

[0134] 因此，在运行期间，虚拟机104第一次使用{1,4,x,10}时，由复值汇编器引用602初始引用的元工厂方法604被执行以创建被高速缓存的部分数组以及上文所描述的工厂方法605。然后用对工厂方法605 的引用来更新复值汇编器引用602。然后虚拟机104执行工厂方法605，接着工厂方法605(例如通过将值从操作数栈402出栈)创建用于x 的当前值的数组并且通过将对该数组的引用推到操作数栈402上来返回该引用。然后发起加载常量指令的方法可以通过例如将引用赋值到局部变量来操纵操作数栈402上的引用。

[0135] 在一些实施例中，在模板化常量的情况中，虚拟机104扩张复值高速缓存引用603以存储多于一个被高速缓存的值。当工厂方法605 实例化新值时，这些值还被高速缓存在

由复值高速缓存引用603指向的位置中。因此，如果相同的变量值被重复使用，则虚拟机104可以返回为这些特定的变量值已经构造的值。

[0136] 在一些实施例中，通过使用模板化常量，与非模板化的技术相比客户代码的大小被减小。例如，通过使用模板化常量，用来创建常量值的虚拟机104指令可以被减小至将变量分量推到操作数栈402上的指令，随后是引用对应的复值条目的加载常量指令。如果使用采用一个或多个分量并且返回用这些分量初始化的数据结构的泛型方法来创建相同的数据结构，则方法可能要求在方法被调用之前所有分量(常量和变量二者)被推到操作数栈402上。因此在一些实施例中，通过使用模板化常量，编译器102仅需要发出将变量值推到操作数栈402 上以替换占位符的指令，从而减小了发出的代码的总大小。

[0137] 在数组方面描述了上面的示例。然而，技术适用于几乎任何种类的值或聚合，诸如复数、组、字典等。

[0138] 作为另一示例，考虑以下复数：

[0139] int x＝….

[0140] ComplexNum a＝x+5i；

[0141] 在实施例中，编译器102在复值条目500中表示复数a，其中复值数据501指定(表示实部的)第一值将由变量供应，而(表示虚部的)第二值位于对存储“5”的整数条目的常量表201的索引处。由复值汇编器指令502表示的元工厂方法604被配置为读取复值数据501 以及实例化/高速缓存ComplexNum数据结构，其中实部是值P，而虚部是值5，其中P是为占位符预留和/或与指示它是占位符的元数据相关联的值。元工厂方法604在被执行时生成工厂方法605，工厂方法 605被配置为读取来自操作数栈402的值以及使用被高速缓存的值的块拷贝和利用从操作数栈402读取的值替换占位符值P的单独拷贝来构造新的ComplexNum。在第一次执行之后，复值汇编器引用602指向所生成的工厂方法605。因此，形式“x+5i”的赋值的将来执行通过利用所生成的工厂方法和由元工厂方法604高速缓存的部分结果来执行。

[0142] 3.4示例编译器过程

[0143] 图7是示出了根据实施例的利用复常量条目创建类文件的示例过程的框图。以下解释假设图7中描绘的过程由编译器102在将源代码文件101转换成类文件103的同时执行。

[0144] 在框700处，编译器102接收指定复常量的使用的源级指令。在实施例中，编译器102接收源代码文件101并且解析包含在源代码文件101内的指令以创建类文件103。在解析源代码文件101时，对于特定指令，编译器102检测指令正在执行赋值或以其他方式加载常量值。在一些实施例中，编译器102存储一组规则，它将这组规则与利用常量值的指令进行比较。例如，编译器102可以了解由虚拟机104 本地支持并且因此可以使用常量表201中的它们各自的条目类型来编码的常量类型。因此，编译器102的规则可以指定已知为本地支持的类型的常量将使用本地类型来编码，而不是被编码为常量表201中的复常量类型。然而，编译器102的规则还可以指定识别不具有已知为虚拟机104本地的类型的常量的具体语法。在这样的情况中，编译器 102将这些常量转换为遵循复值条目500的格式的条目。由编译器102 支持的确切的复常量是实施方式特定的，但是本文所描述的技术可以被应用于编译器102的开发者希望支持的几乎任何类型的复常量。

[0145] 例如，编译器102可以被配置为识别复数常量并且将这些常量转换为类文件200中的复值条目。例如，语法“ComplexNum＝6+9i”可以指示ComplexNum应当被设置为引用实部

是“6”、虚部是“9i”的复数的实例。

[0146] 作为另一示例，编译器102可以被配置为识别组常量并且将这些常量组转换为类文件200中的复值条目。例如，语法“G＝{1,true, “hello”}”可以指示G应当被设置为引用包含整数“1”、布尔值“真 (true)”和字符串“hello”的组的实例。

[0147] 作为又一示例，编译器102可以被配置为识别部分常量以作为用于上文在节3.3中所描述的模板化常量的候选者使用。例如，对于由编译器102支持的任何复常量，遵循适当语法、但是使用变量来供应常量的一部分的指令可以被识别为用于模板化常量的候选者。

[0148] 在框701处，编译器102确定创建复常量的存储技术和汇编器代码。在实施例中，编译器102基于诸如元素的数量、所需要的空间、常量的类型等之类的复常量的因素来确定存储技术。例如，如果存储复值数据501所需要的空间在特定阈值之下并且分量部分是使用本地类型可表示的，则编译器102可以为这些分量部分在常量表201中生成附加条目并且在复值数据501中存储对这些条目的索引。否则，编译器102可以生成用于常量表201的“字节”条目以用作数据的原始容器并且将条目存储到复值数据501的“字节”条目。可替代地，一些实施例可以直接在复值数据501中存储用于生成复常量的值的数据。

[0149] 在实施例中，编译器102存储将语法的特定实例映射到用于处理复常量的相关联的类型的元工厂代码的一个或多个规则。因此，在这样的实施例中，编译器102使用一个或多个规则识别用于特定指令语法的常量类型以及确定要在复值汇编器指令502中放置或引用的元工厂代码。在一些实施例中，为每个复常量类型定义多个元工厂以处理用于该复类型的数据可以存储的各种格式。例如，一个元工厂可以被配置为处理复值数据501对于常量的分量部分引用本地常量表201条目的情况，而另一个元工厂可以被配置为处理复值数据501引用包含数据的原始字节的字节条目的情况。

[0150] 例如，在复数的情况中，编译器102可以选择在常量表201中生成表示复数的实部和虚部的两个整数条目。因此，当生成用于复数的复值条目500时，复值数据501存储对两个整数条目的引用。此外，编译器102基于指令的语法将常量类型识别为复数并且在复值汇编器指令502中存储或引用元工厂，该元工厂被配置为当数据被存储在两个整数条目中时处理复数。

[0151] 作为另一示例，在组的情况中，编译器102可以选择在常量表201 中生成存储用于组的元素的原始数据的字节条目。因此，当生成用于复数的复值条目500时，复值数据501存储对字节条目的引用。此外，编译器102基于指令的语法将常量类型识别为组并且在复值汇编器指令502中存储或引用元工厂，该元工厂被配置为当数据被存储在原始字节容器中时处理组。

[0152] 在框702处，编译器102基于确定的存储技术和汇编器代码来生成一个或多个常量表条目。在实施例中，编译器102通过在复值数据 501中存储数据或对数据的引用以及在复值汇编器指令502中存储元工厂方法或对元工厂方法的引用来生成用于复常量的复值条目500。此外，取决于所选择的存储技术，编译器102可以生成将由复值数据 501引用的常量表201的其他条目，诸如“字节”条目或表示虚拟机 104的本地类型的条目。

[0153] 在一些实施例中，对于模板化常量，编译器102被配置为生成用于使用模板化常量的方法的附加指令，该方法将变量的值推到操作数栈402上，工厂方法可以从操作数栈402获得这些值。

[0154] 3.5示例虚拟机过程

[0155] 图8是示出了根据实施例的用于加载复常量的示例过程的框图。在图8中，假设过程由虚拟机104在执行由类文件103表示的程序的同时被执行。附加地，为了避免不必要地使示例复杂，上文关于加载、链接和初始化过程所讨论的许多步骤从解释中被省略。下面的示例假设用于利用复常量的类的类文件200已经被加载到由虚拟机存储器布局300表示的运行时存储器中。此外，示例假设虚拟机104经由解释器108解释程序，但是技术同样适用于程序或程序的部分在执行前经由JIT编译器109编译的情况。

[0156] 在框800处，虚拟机104接收引用运行时常量池304的运行时复常量条目600的指令。在实施例中，虚拟机104接收采取对运行时常量池304的索引作为输入并且将条目引用的常量的值推到操作数栈 402上的指令。例如，为了将来自运行时常量池304的值赋值到局部变量，包括该方法的虚拟机指令可以包括加载常量的指令以及将该常量赋值到局部变量的另一指令。例如，在JVM的背景下，这将被表示为引用运行时常量池304的条目的ldc指令，随后是指示局部变量 401内的局部变量的索引的astore指令。在该情况中，假设由虚拟机104接收的指令引用的运行时常量池304条目是运行时复常量条目 600。

[0157] 在实施例中，虚拟机104的加载常量指令被重载以对于多个类型的常量起作用。因此，例如虚拟机104可以查询与所引用的运行时常量池304条目相关联的元数据以确定常量的类型。然后加载常量指令遵循用于加载该特定类型的常量的执行路径。然而，在其他实施例中，虚拟机104支持多个指令，每个指令适合不同类型或类别的常量。例如，用于本地常量的指令、用于复常量的指令、用于模板化的复常量的指令等等。

[0158] 在框801处，虚拟机104确定运行时复常量条目600是否具有被高速缓存的值。如果运行时复常量条目600具有被高速缓存的值，则虚拟机104前进到框802。如果运行时复常量条目600不具有被高速缓存的值，则虚拟机104前进到框804。在实施例中，虚拟机104检查运行时复常量条目600的复值高速缓存引用603以确定运行时复常量条目600是否包含对被高速缓存的值的引用。如果这是复常量第一次被利用，则复值高速缓存引用603将为空。否则，复值高速缓存引用603将指向被高速缓存的值的位置，在该示例中该位置位于堆302 上。

[0159] 在框802处，虚拟机104执行由复值汇编器引用602引用的元工厂方法以生成工厂方法605和用于复常量的值。在实施例中，虚拟机 104执行由运行时复常量条目600的复值汇编器引用602初始引用的元工厂方法604。元工厂方法604读取运行时复值数据601并且构造用于复常量的值。例如，元工厂方法可以在堆302上为该值分配空间并且然后用来自运行时复值数据601的值来填充该空间。附加地，元工厂方法604生成工厂方法605，工厂方法605返回由元工厂方法604 构造的值。在实现模板化常量的实施例中，如果复常量是模板化常量，则由元工厂方法604构造的值包含将由一个或多个变量供应的一个或多个占位符值。此外，在这样的情况中，工厂方法605被配置为将由元工厂方法604生成的值与(例如，来自操作数栈402的)一个或多个变量值组合以创建(例如，通过将对新值的引用推到操作数栈402上)返回到加载常量的方法的新值。

[0160] 在框803处，虚拟机104存储生成的值并且将复值汇编器引用602 更新为引用所生成的工厂方法605。在实施例中，虚拟机104将复值高速缓存引用603更新为指向所生成的值并且将复值汇编器引用602 更新为指向工厂方法605。

[0161] 在框804处，虚拟机执行工厂方法605以返回复常量的值。在实施例中，工厂方法

605通过将对该值的引用推到操作数栈402上来返回该值。然而，在模板化常量的情况中，工厂方法605(例如，在堆 302上)分配用于新值的空间、执行由复值高速缓存引用603引用的值到新值的位置的块拷贝，然后执行附加的拷贝以利用从操作数栈 402出栈的变量值来替换占位符值。然后工厂方法605例如通过将新值的存储器位置的存储器地址推到操作数栈402上来返回对该位置的引用。然后加载复常量的方法可以对存储在操作数栈402上的值执行操作，诸如将该值加载到局部变量中。

[0162] 在一些实施例中，在框800处引用运行时复常量条目600的指令包括影响工厂方法605返回复常量的值的方式的一个或多个附加参量。例如，引用运行时复常量条目600的指令可以指示返回值应当是可变的。在一些实施例中，在这样的情况中，工厂方法总是在可变的数据位置中存储复常量的值，甚至在非模板化常量的情况中也是如此。因此，加载复常量的方法可以转变位于该引用处的值，而不存在与其他变量的交叉影响的风险。

[0163] 3.6引用vs值类型

[0164] 对于诸如Java编程语言之类的一些语言，不是原生的类型替代地是引用类型。引用类型通过操作数栈402上的引用(作为引用入栈和出栈)传递并且被存储为诸如其他Object之类的容器中的引用。加载常量的过程的先前解释假设生成的常量值是引用类型。然而，在其他实施例中，复常量类型可以被当作原生。

[0165] 在一些实施例中，作为通过引用传递复常量的值的替代，复常量通过值来传递。例如，由工厂方法605推到操作数栈402上/从操作数栈402出栈或者放置在局部变量401内或者包含Object的复常量值通过值而不是通过引用而被实现。此外，在一些实施例中，虚拟机104 基于复常量的大小和复杂度在通过值传递和通过引用传递语义之间自由地切换。对于元素的数量和/或大小在特定阈值之下的较简单的复常量，虚拟机104使用通过值传递语义。然而，在元素的数量和/或大小在特定阈值之上的情况中，虚拟机104使用通过引用传递语义。在一些情况中，当复值相对小并且易于容纳在操作数栈402和/或局部变量 401中时，通过值传递语义是优选的，由于它移除了如果值通过引用来传递的话否则将被执行的某一级别的间接引用(indirection)。然而，当复常量具有太多分量或太大时，将复常量的值复制到操作数栈 402/从操作数栈402复制复常量的值的行为移除间接引用的优点被复制的开销掩盖。在一些实施例中，虚拟机104提供用于如由John R. Rose等人于2015年4月29日提交的、名称为“Handling Value Types (处理值类型)”的、申请号为14/699,129的美国专利申请中所描述的值类型的复常量，在此出于所有目的通过引用将该美国专利申请的全部内容并入本文，如同在此完全阐述一样。

[0166] 3.7常量的可组合性

[0167] 在一些实施例中，复常量是可组合的，这意味着复常量可以将其他复常量用作复值数据501的部分，其中值从复值数据501生成。例如，作为引用表示本地支持的类型的其他常量表201条目的附加或替代，复值数据501可以引用表示其他复值条目的其他常量表201条目。在这样的实施例中，加载取决于其他复常量的复常量引发加载的递归链。

[0168] 作为示例，考虑复常量G([1,2,3],5,“Hello World”)的创建，该复常量是由数组[1,2,3]、整数[5]和字符串“Hello World”组成的组。在用于G的运行时复值条目600中，运行时复值数据601引用运行时常量池303的三个条目，第一个条目“A”是引用用于“1”、“2”和“3”的整数条目的另一复值条目，第二条目“B”是用于“5”的整数条目，而第三条目“C”是用

于字符串“Hello World”的字符串条目。附加地，假设这是G的第一次加载，则用于G的运行时复值条目 600包括复值汇编器引用602，复值汇编器引用602初始指向元工厂方法604，元工厂方法604加载条目A、条目B和条目C、生成组G、将得到的组高速缓存、产生通过将对被高速缓存的组的引用推到操作数栈402上来返回组数据结构的工厂方法605、将复值汇编器引用602 更新为引用所产生的工厂方法605以及执行工厂方法605。

[0169] 为了加载条目A，运行环境113递归执行由条目A引用的方法。如果这是第一次加载由条目A表示的复常量，则引用将引出从指定的整数条目生成数组[1,2,3]以及构造/执行工厂方法的元工厂方法，否则引用将引出经由操作数栈402返回被高速缓存的结果的工厂方法。在任一情况中，执行路径将引出对被放置在操作数栈402上以供由用于 G的运行时复值条目600的元工厂方法604使用的数组[1,2,3]的引用。

[0170] 尽管之前的示例描述了仅两层的复常量，但是由于复常量建立在彼此之上，因此其他实施例可以包含几乎任何数量的复常量层。此外，可组合的常量与上文在节3.3中描述的模板化常量机制兼容。例如，较高级别的工厂方法可以将一个或多个变量值向下传递到下一个较低级别的工厂方法以在生成值时使用。

[0171] 4.0硬件概述

[0172] 根据一个实施例，本文描述的技术由一个或多个专用计算设备实现。专用计算设备可以被硬连线以执行技术，或者可以包括被永久编程以执行技术的诸如一个或多个专用集成电路(ASIC)或现场可编程门阵列(FPGA)之类的数字电子设备，或者可以包括被编程以根据固件、存储器、其他存储或者组合中的程序指令来执行技术的一个或多个通用硬件处理器。这样的专用计算设备还可以用定制编程将定制的硬连线逻辑、ASIC或FPGA组合以实现技术。专用计算设备可以是将硬连线逻辑和/或程序逻辑结合以实现技术的桌上型计算机系统、便携式计算机系统、手持设备、网络(networking)设备或任何其他设备。

[0173] 例如，图9是示出在其上可以实现本发明的实施例的计算机系统 900的框图。计算机系统900包括用于传送信息的总线902或其他通信机构，以及用于处理信息的、与总线902耦接的硬件处理器904。硬件处理器904可以是例如通用微处理器。

[0174] 计算机系统900还包括用于存储将由处理器904执行的信息和指令的、耦接到总线902的主存储器906，诸如随机存取存储器(RAM) 或其他动态存储设备。主存储器906还可以被用于在由处理器904执行的指令的执行期间存储临时变量或其他中间信息。这样的指令当被存储在对于处理器904来说可访问的非暂态存储媒介中时，使计算机系统900变为被定制以执行指令中指定的操作的专用机器。

[0175] 计算机系统900还包括用于为处理器904存储静态信息和指令的、耦接到总线902的只读存储器(ROM)908或其他静态存储设备。诸如磁盘、光盘或固态驱动器之类的存储设备910被提供并且被耦接到总线902以用于存储信息和指令。

[0176] 计算机系统900可以经由总线902被耦接到诸如发光二极管 (LED)显示器之类的显示器912以用于向计算机用户显示信息。包括字母数字和其他键的输入设备914被耦接到总线902以用于向处理器904传送信息和命令选择。另一种类型的用户输入设备是用于向处理器904传送方向信息和命令选择以及用于控制显示器912上的光标移动的光标控制器916，诸如鼠标、跟踪球或光标方向键。该输入设备通常具有允许设备在平面中指定位置的在两个轴——第一轴(例如， x)和第二轴(例如，y)上的两个自由度。

[0177] 计算机系统900可以使用与计算机系统结合使得计算机系统900 成为专用机器或者将计算机系统900编程为专用机器的定制的硬连线逻辑、一个或多个ASIC或FPGA、固件和/或程序逻辑来实现本文描述的技术。根据一个实施例，响应于处理器904执行包含在主存储器 906中的一个或多个指令的一个或多个序列，本文的技术由计算机系统900执行。这些指令可以从诸如存储设备910之类的另一个存储介质被读取到主存储器906中。包含在主存储器906中的指令的序列的执行使得处理器904执行本文所描述的过程步骤。在可替代的实施例中，硬连线电路可以代替软件指令或与软件指令结合使用。

[0178] 如在此所使用的术语“存储媒介”是指存储使得机器以特定方式操作的数据和/或指令的任何非暂态媒介。这样的存储媒介可以包括非易失性媒介和/或易失性媒介。非易失性媒介包括例如光盘、磁盘或固态驱动器，诸如存储设备910。易失性媒介包括动态存储器，诸如主存储器906。常见形式的存储媒介包括例如软盘、柔性盘、硬盘、固态驱动器、磁带或任何其他磁数据存储介质、CD-ROM、任何其他光数据存储介质、具有孔的图案的任何物理介质、RAM、PROM和 EPROM、FLASH-EPROM、NVRAM、任何其他存储器芯片或记忆卡。

[0179] 存储媒介不同于传输媒介，但是可以结合传输媒介被使用。传输媒介参与在存储媒介之间传送信息。例如，传输媒介包括同轴线缆、铜线和光纤，这些传输媒介包含包括总线402的线。传输媒介还可以采取声波或光波的形式，诸如在无线电波和红外数据通信期间生成的那些声波或光波。

[0180] 将一个或多个指令的一个或多个序列承载到处理器904以用于执行可以涉及各种形式的媒介。例如，指令可以初始地被承载在远程计算机的磁盘或固态驱动器上。远程计算机可以将指令加载到它的动态存储器中并且使用调制解调器通过电话线发送指令。计算机系统900 的本地调制解调器可以接收电话线上的数据并且使用红外传输器将数据转换成红外信号。红外检测器可以接收红外信号中承载的数据而合适的电路可以将数据放置在总线902上。总线902将数据承载到主存储器906，处理器904从主存储器906检索指令并且执行指令。由主存储器906接收的指令可以可选地或者在由处理器904执行之前或者在由处理器904执行之后被存储在存储设备910上。

[0181] 计算机系统900还包括耦接到总线902的通信接口918。通信接口918提供耦接到网络链路920的双向数据通信，网络链路920被连接到本地网络922。例如，通信接口918可以是综合业务数字网(ISDN) 卡、线缆调制解调器、卫星调制解调器或者调制解调器以提供到对应类型的电话线的数据通信连接。作为另一示例，通信接口918可以是局域网(LAN)卡以提供到兼容的LAN的数据通信连接。无线链路也可以被实现。在任何这样的实施方式中，通信接口918发送和接收承载表示各种类型的信息的数字数据流的电信号、电磁信号或光信号。

[0182] 网络链路920通常通过一个或多个网络提供到其他数据设备的数据通信。例如，网络链路920可以通过本地网络922提供到主机计算机924或到由网络服务提供商(ISP)926操作的数据装置的连接。ISP 926又通过现在通常被称为“因特网”928的全球分组数据通信网提供数据通信服务。本地网络922和因特网928都使用承载数字数据流的电信号、电磁信号或光信号。通过各种网络的信号和在网络链路920 上并通过通信接口918的信号是传输媒介的示例形式，所述信号承载到计算机系统900的数字数据以及来自计算机系统900的数字数据。

[0183] 计算机系统900可以通过网络(多个网络)、网络链路920和通信接口918来发送消

息和接收包括程序代码的数据。在因特网示例中，服务器930可以通过因特网928、ISP 926、本地网络922和通信接口 918来传送用于应用程序的请求的代码。

[0184] 接收的代码可以在被接收时由处理器904执行，和/或存储在存储设备910或者其他非易失性存储中以用于以后执行。

[0185] 如本文所使用的，术语“一个”、“两个”、“某个”和“特定的”被用作命名约定以将查询、计划、表示、步骤、对象、设备或其他项彼此区分开，以使得这些项在它们被引入之后可以被引用。除非在此另外指定，这些术语的使用不暗示所引用的项的顺序、时序或任何其他特性。

[0186] 5.0扩展和替代物

[0187] 在上述说明书中，参考可以根据实施方式而不同的许多具体细节描述了本发明的实施例。因此，本发明是什么以及申请人意图将什么作为本发明的唯一且排他的指示物是以权利要求发布的具体形式从本申请中发布的一组权利要求，包括任何后续补正。用于包含在权利要求中的术语的本文明确阐述的任何定义应当决定在权利要求中使用的这些术语的含义。因此，在权利要求中没有明确表述的限制、元素、性质、特征、优点或属性都不应当以任何方式限制这些权利要求的范围。因此，说明书和附图应被视为是说明意义上的而不是限制意义上的。

[0188] 6.0第一附加公开

[0189] 本文描述了用于组织(structure)和利用与程序数据相联系的静态数据区域的技术，以便于促进数组常量和其他复常量的使用以及用于其他目的。

[0190] 根据实施例，一种方法包括在一组指令内识别加载数据的指令。例如，诸如Java虚拟机(“JVM”)之类的解释器可以解释较低级别的编译后的指令(例如，非人类可读的源代码)，诸如Java字节码。解释器可以遇到将数据加载到诸如操作数栈、寄存器、堆等之类的运行时数据区域的指令。在基于Java编程环境的实施例中，这样的指令的示例可以包括“ldc”或“invokedynamic”。

[0191] 该方法还包括识别静态数据区域内的第一条目，该第一条目与指令相关联。例如，一组指令可以具有程序数据的一个或多个相关联的只读区域，诸如当这组指令被执行时在同一Java.class文件内的常量池。指令可以指定用于遵循指令或者以其他方式与指令相关联的参量中的条目的地址。

[0192] 方法还包括识别第一条目指定信息，该信息至少指示：(a)方法指定信息，该方法指定信息引用或指定被配置为指引数组结构的生成的方法代码的位置；以及(b)值规范信息，该值规范信息指定要在数组结构中包括的值。在实施例中，为了支持识别过程，第一条目可以具有指定的类型(例如，预定义的常量池条目类型，诸如 CONSTANT_Dynamic)。指定信息可以直接和/或通过对其他结构(诸如对静态数据区域中的另一条目和/或对引导(bootstrap)属性结构中的条目)的显式或隐式引用(多个引用)来指示(a)和(b)。方法指定信息可以引用各种形式的适合的方法代码，诸如，例如实现被配置为生成和/或返回被配置为生成数组结构的附加方法代码的元工厂的方法代码、由工厂实现的方法代码、用于被配置为生成数组结构的引导方法的方法代码、用于对于被配置为生成数组结构的方法的方法句柄 (handle)的方法代码等等。值指定信息可以是例如值的序列、对各自单独存储值的、静态数据区域中的其他条目的引用的序列和/或对存储表示一组值的比特的、静态数据区

域中的其他条目(多个条目)的引用(多个引用)。

[0193] 指定信息可选地还指示参量指定信息,该参量指定信息指定被配置为影响数组结构将被生成的方式的、要传递到方法代码的一个或多个参量,诸如静态参数。例如,参量指定信息可以指定数组结构中的值的类型(多个类型)、数组结构中的值是否将是不可变的等等。在实施例中,这些特性中的一些或全部可以以数组逐个索引的粒度被指示,以使得某些被索引的元素可以具有一种类型,而其他被索引的元素可以具有另一类型。在数组将由元工厂产生的附加方法代码生成的实施例中,一个或多个参量是被传递到元工厂的静态参量,并且元工厂被配置为基于一个或多个参量的值来生成不同的数组形成的方法代码。由元工厂产生的附加方法代码可以自身被配置为在运行时采取影响数组生成的动态参量。在数组将由方法指定信息中指示的所引用的方法代码直接生成的实施例中,所引用的方法代码包括取决于不同参量的值来生成不同数组的逻辑。

[0194] 方法还包括使用值以及可选地使用参量来执行方法代码以生成数组结构。可选地,取决于指令,可以利用数组结构执行各种动作,诸如基于该组指令中的另一指令将数组结构推到栈或其他可写的运行时数据区域上,以及使数组结构从栈出栈或从运行时数据区域弹出。

[0195] 在实施例中,指定信息还指示数组的至少一个值将从动态数据区域(例如,操作数栈、寄存器等)检索。因此,数组被视为半常量数组或模板,其中某些元素是静态的,而其他元素根据程序数据的执行被动态填充。

[0196] 根据实施例,另一方法包括在静态数据区域内识别第一条目,该第一条目包括引用静态数据区域中的第二条目的特定属性。方法还包括确定第二条目具有对应于一组常量值的数据类型。例如,第二条目可以具有指示它对应于一组常量值的指定类型(例如,预定义的常量池条目类型,诸如CONSTANT_Group)。方法还包括识别第二条目内的多个属性,该多个属性中的每个属性引用从其找到对应于该属性的值的静态数据区域中的不同条目。方法还包括用多个属性取代特定属性。

[0197] 在实施例中,多个属性对应于要包括在数组中的常量值。例如,多个属性可以对应于由上面的值指定信息指定的值,并且因此被用于生成诸如上面生成的数组结构之类的数组结构。

[0198] 在实施例中,多个属性中的第二特定属性指代对应于另一组常量值的静态数据区域的第三条目。因此,通过条目链,任何数量的属性可以被引用。

[0199] 根据实施例,另一方法包括在静态数据区域内识别第一条目,该第一条目包括引用静态数据区域中的第二条目的特定属性。方法还包括确定第二条目具有对应于一组常量值的特定数据类型。例如,第二条目可以具有指示它包含可以被解析以产生该组常量值的字节的序列的指定类型(例如,预定义的常量池条目类型,诸如 CONSTANT_Group)。方法还包括将第二条目中的字节序列传递到方法代码。例如,字节序列可以作为被传递到上文所描述的数组生成方法代码或被传递到任何其他引导方法中的单个属性被传递。方法代码被配置为解析字节序列以及将字节序列解释为值序列。以这种方式,多个值在静态数据区域的单个条目而不是一组条目内被有效地直接存储。

[0200] 在实施例中,值序列至少是由上文所描述的值指定信息指定的值的子集。

[0201] 在实施例中,解释器(例如,执行引用来自静态数据区域的条目的一组指令的组

件)通过其解释特定数据类型的条目的逻辑被动态链接。因此根据由于条目被传递到的方法代码和/或由于被传递到方法代码的其他属性(诸如数组值类型信息)的不同逻辑来解释具有相同特定数据类型(例如,CONSTANT_Byte)的不同条目中的字节序列。

[0202] 在实施例中,一种方法包括响应于识别创建包括静态值的数组的较高级别的指令来生成包括上文所描述的指令中的任何一个或多个指令的较低级别的指令(多个指令),以及生成包括上文所描述的条目中的任何一个或多个条目的静态数据区域。例如,编译器可以响应于在将(例如,以Java编程语言的)人类可读的源代码编译为诸如Java 字节码之类的较低级别的一组指令时识别常量数组声明来执行该方法。

[0203] 取决于实施例和编程环境,在此指代的“静态数据区域”可以采取各种形式。在一些实施例中,静态数据区域是潜在地结合由常量池中的条目可寻址的Java.class文件内的若干辅助静态数据结构的 Java.class文件的常量池。作为常量池的替代,其他实施例可以利用用于程序数据的任何只读存储器区域(而不是用于程序数据的可写存储器区域)。这样的存储器区域可以是例如从被执行的一组指令分开、但是由这组指令可寻址的固定数据的汇集。作为另一示例,这样的存储器区域可以是由被执行的一组指令引用的文本/常量值的有序集合。在实施例中,静态数据区域可以是已经(经由编译、解释或以其他方式)准备用于在使用或以其他方式引用来自数据的值的指令的执行期间由虚拟机或解释器使用的任何数据。

[0204] 在实施例中,常量池可以被用于生成常量数组和/或其他复常量,而不必向常量池添加将条目识别为常量数组的新指令。为了该目的,利用有关常量池的解释器(诸如Java虚拟机)的现有功能。条目通过引用元工厂或少数元工厂中的一个来“表示”复常量,而不是将绑定到(tied to)常量池中的特定类型的条目的固定常量数组生成逻辑添加到解释器。元工厂则可以被配置为基于常量池中的静态参量(多个静态参量)生成常量数组生成逻辑(或复常量生成逻辑)的不同实现。该实施例提供了容易适应于新类型的复常量的附加灵活性,以便于避免需要添加若干附加常量池指令来为访问为了支持各种各样的常量数组或复常量类型所需要的各种各样的复常量生成逻辑提供便利。当然,在其他实施例中,常量池可以包括具体对应到生成常量数组或其他复常量的指令的一个或多个类型的条目。

[0205] 虽然本文关于Java常量池具体描述了某些技术,但是要理解的是常量池仅是关于其可以实践本文描述的技术的示例类型的存储器区域,而在其他实施例中技术同样适用于上文所描述的静态数据区域中的任何静态数据区域。

[0206] 在其他方面,公开了用于实现本文所描述的技术的计算设备和计算机可读媒介的系统。

[0207] 8.0第二附加公开

[0208] 根据实施例,本文所描述的技术解决的一个问题是支持常量池中的数组常量的问题。在其他实施例中,本文所描述的技术适合于解决一系列其他问题,诸如任意常量类型,以及甚至半常量模板。

[0209] 用于程序数据的常规常量池和其他只读存储器区域具有表示小范围的常量值(包括原生、类、字符串和方法句柄)的本地能力。将这些能力匹配到全范围的可能的冻结数组常量是有挑战的。例如,以下Java表达式是有问题的:

[0210] final Object final[]THREE_THINGS=＝{false,'1',short.class};

[0211] 常量表达式“false”转换成“Boolean.FALSE”，并且这两个值在常量池中都不是直接可表示的。用于数字“1”(49)的代码点在常量池中可以被表示为整型(尽管它因具有五个字节而相当庞大)，但是没有地方可以找到装箱值“Character.valueOf('1')”。尽管“CONSTANT_Class”常量可以对许多类进行编码，但是它们(目前) 不能表达用于“短整型”的原生“类”对象。

[0212] 为了简化实现，可以将数组常量限制为原生值类型的数组。但是即使这样也存在问题。不管数组的实际动态范围多大，“长整型”的数组的朴素表示将每元素在类文件中消耗(burn)64比特。静态类大小可以影响启动性能，而我们不会通过使静态类大小不必要地庞大而驱使用户离开新机制。

[0213] 在这点处，“invokedynamic”提供了良好的替代物。用于数组常量(以及任何其他常量!)的合适范围的静态表示是使用某一范围的有用的元工厂可产生的，这些元工厂中的每个可以链接产生常量的“invokedynamic”调用站点。传递到每个元工厂的静态元数据可以是 (几乎)来自常量池的常量的任何序列。运行时库，而不是JVM，可以定义数组常量元工厂的该范围。这将表示问题从JVM及其类文件格式向上推到库。该方案更加不会过时，并且可以在没有新常量池常量的情况下表示将来类型(类似扁平值数组常量)。

[0214] 在实施例中，可以通过仅添加两种常量条目池类型(即，常量池中的给定条目遵从的数据的类型，有时在此简称为“常量类型”)来支持实施方式：一种是原始比特的源，一种是具有将常量池常量组装成常量的功能的一组常量池常量。出乎意料地，这两种常量池类型都不映射到调用“CONSTANT_Array”将会希望的类型。

[0215] “CONSTANT_Dynamic”(新常量标签17)将解析为对指定集合的参数执行引导方法(元工厂)的结果。“CONSTANT_Dynamic”的第一参量将是引导说明符(specifier)，它与“CONSTANT_InvokeDynamic”的第一分量相同。第二参量将是“CONSTANT_Class”，它给出常量的期望类型。(第一次)解析该常量将执行引导说明符。作为(如“invokedynamic”指令要求的)返回“CallSite”对象的替代，引导方法将返回值，该值将立即转换成要求的类型。

[0216] 在实施例中，第二操作数可以是从引导方法的返回类型可导出的，但是在其他实施例中它不应当被省略，因为可能存在较小的有限数量的元工厂，但是存在无限数量的常量类型。附加地，如果常量类型是立即可见的，则验证和结果追踪更简单。比较执行类似功能的“getfield”调用中的“type”操作数。

[0217] “CONSTANT_Bytes”常量(新常量标签2)将创建一系列字节的(在“CharSequence”上建模的类型“ByteSequence”的)只读视图。字节将立即跟随最前的“u4”大小字段。在实施例中，类文件解析器和JVM将根本不解析这些字节，但是允许引导方法按期望处理它们。

[0218] 通过使用这两种常量类型，加上元工厂(引导方法)的精选集合，大多数(不是全部)种类的常量可以被计算。

[0219] 在实施例中，不存在将字节显现为冻结的字节数组的调用，因为这将要求某些JVM实现执行从类文件到堆中的额外拷贝。

[0220] 在实施例中，可以将“CONSTANT_Dynamic”压缩(collapse) 成“CONSTANT_InvokeDynamic”。然而，在其他实施例中，这没有帮助。例如，“CONSTANT_InvokeDynamic”不能是“ldc”的操作数并且因此不能自身当作引导方法参数。在实施例中，为了打破用于引导方法的参数个数限制，对于一些用例，嵌套的引导方法参数是必需的。

[0221] 在实施例中,在某些实施例中由该方案可能引起的问题是某些 JVM具有作为对引导方法可用的操作数的最大数量的大约250个常量池引用的相当奇怪的限制。但是这可以通过将多层“CONSTANT_Dynamic”常量入栈而被解决。

[0222] 在实施例中,“CONSTANT_ConstantGroup”常量类型也可以是有用的。该常量类型可以表示要被显现为元工厂将读取并且打包到期望的数组中的类似于“ByteSequence”句柄的“ConstantSequence”句柄的常量值的很长的序列(在一些实施例中数千个或者甚至数百万个)。该序列将不一定是主常量池的部分,但是可以是类文件内的附带的类似池的结构。然而,在其他实施例中,由于“BootstrapSpecifiers”属性已经具有相似的结构,因此使用该预先存在的结构而不是创建新的、稍微不同的结构可以是有用的。

[0223] 示例

[0224] 在实施例中,用于指定数组的示例常量池条目可以如下所示。在示例中,AMF引用一般用于生成数组的假设的元工厂。BAMF表示为了生成专有类型的数组(例如,布尔值的数组)而优化的假设的元工厂。尽管在一些实施例中这样的专有元工厂可以存在,但是将认识到在其他实施例中它们不是严格必需的。

[0225] AMF

[0226] int/specifying data/

[0227] 1

[0228] 5

[0229] 3

[0230] 常量池中的条目的上面的示例可以被用于使用由AMF元工厂产生的方法代码来生成整型的数组。“int”和“/specifying data/”二者作为静态参数被传递到所引用的元工厂,并且影响由AMF为了生成数组而产生的方法代码。“整型(int)”是指定数据的示例,它指示在数组中发现的元素的类型。更复杂的数组可以具有各自附属于数组内的具体的、被索引的元素的多个数据类型。另一“/specifying data/”可以包括影响数组生成的任何数量的参量,诸如数组是否将是不可变的,数组内的元素怎样存储在常量池(或其他存储器区域)中等等。列表中的最后三个条目是形成数组{1,5,3}的数组元素的实际值。

[0231] 尽管上文示出的条目被顺序列出,但是将认识到在各种实施例中它们可以被散布在常量池或类文件内的任何其他合适的数据区域内的各处。第一条目可以包括引用第二条目(或者引出或集体产生第二条目的条目链)的位置的参量等。或者在其他实施例中,指定数据可以与对元工厂的引用组合。

[0232] 其他示例如下所示。在每个示例中,每个条目由逗号分开。字母 G表示包含对括号中列出的一组条目的引用的单个条目(例如,被指定为CONSTANT_Group)。字母D表示包含(如括号内所描绘的) 字节的单个条目,当这些字节由AMF元工厂生成的方法解析时,产生用于数组的一列值。“b[]”指代指定信息,该指定信息在该情况中指示二进制数组。

[0233] AMF,b[],1,0,1

[0234] BAMF,1,0,1

[0235] BAMF,G(1,0,1)

[0236] BAMF,D(05)

[0237] BAMF,G(int,1,0,1)

[0238] 在实施例中,可以产生半常量数组或模板。例如,考虑以下代码:

[0239] int x=…

[0240] int[]a={1,4,x,10};

[0241] 该代码可以被编译成诸如以下字节码之类的字节码。

[0242] aload#x

[0243] invokedynamic TAMF#123

[0244] astore#a

[0245] 在该示例中,“TAMF”指代用于生成模板数组的元工厂。常量池条目#123如下:

[0246] G(1,4,-,10)

[0247] 其中-是用于其值将动态供应(例如,从堆弹出或者从寄存器或局部变量供应)的数组元素的符号或其他合适的表示。可替代地或附加地,指定信息可以作为调用的属性被供应,该调用的属性指示其值将被动态供应的数组中的元素的索引(多个索引)的列表。

[0248] 在实施例中,响应于该字节码,所引用的TAMF元工厂可以生成代码,诸如:

[0249] mf=λ(z){return new int(){1,4,z,10}}

[0250] 9.0第三附加公开

[0251] 本文所描述的主题的方面列举在以下标号的条款中:

[0252] 1.一种方法,包括:在一组程序指令内识别加载常量的指令;基于加载常量的指令识别与识别由程序使用的一个或多个常量的数据结构中的常量相关联的第一条目,其中第一条目至少指定常量数据以及用于从常量数据组装值或部分值的第一组指令;执行第一组指令以从常量数据组装值或部分值;将特定值或对特定值的引用存储到被用于在运行环境中执行的指令组之间传递值或引用的运行时数据结构上,其中特定值基于从常量数据组装的值或部分值,其中方法由一个或多个计算设备执行。

[0253] 2.条款1的方法,其中常量表示数组、组、序列、字典、复数、点或对象中的一个或多个。

[0254] 3.条款1-2中的任一条款的方法,其中识别由程序使用的一个或多个常量的数据结构是包含多个条目的常量表,其中多个条目中的每个条目与不同常量相关。

[0255] 4.条款1的方法,其中执行第一组指令在常量值高速缓存中存储值或部分值。

[0256] 5.条款4的方法,其中存储在常量值高速缓存中的值或部分值被存储在值或部分值不可变的存储器位置处。

[0257] 6.条款4的方法,其中执行第一组指令生成第二组指令,当第二组指令被执行时,基于存储在常量值高速缓存中的值或部分值生成特定值,并且执行将特定值或对特定值的引用存储到运行时数据结构上。

[0258] 7.条款6的方法,其中数据结构中的第一条目包括指向第一组指令的汇编器引用,并且执行第一组指令使得汇编器引用被更新为指向第二组指令。

[0259] 8.条款7的方法,还包括:

[0260] 在第二组程序指令内识别加载常量的第二指令;基于加载常量的第二指令识别与识别由程序使用的一个或多个常量的数据结构中的常量相关联的第一条目,其中第一条目至少识别常量值高速缓存中的值或部分值和第二组指令;执行第二组指令以基于常量值高速缓存中的值或部分值将第二特定值或对第二特定值的引用存储到运行时数据结构上。

[0261] 9.条款6-8中的任一条款的方法，其中执行第一组指令组装部分值，其中部分值是包含一个或多个常量分量以及一个或多个变量分量的聚合，其中第二组指令通过执行从常量值高速缓存到表示特定值的运行时存储器中的分开的位置的部分值的块拷贝以及为了替换一个或多个变量分量的、来自运行时数据结构的变量值的一个或多个拷贝来生成特定值。

[0262] 10.条款1-9中的任一条款的方法，其中将特定值或对特定值的引用存储到运行时数据结构上将对特定值的引用存储到运行时数据结构上。

[0263] 11.条款1-10中的任一条款的方法，其中加载常量的指令包括一个或多个参量，该一个或多个参量确定对特定值的引用是否指向运行时存储器中的可变位置。

[0264] 12.一个或多个非暂态计算机可读媒介，该一个或多个非暂态计算机可读媒介存储指令，当指令由一个或多个计算设备执行时，使得条款1-11中所述的方法中的任何一种方法被执行。

[0265] 13.一种系统，该系统包括一个或多个计算设备，该一个或多个计算设备包括至少部分地由计算硬件实现的组件，所述组件被配置为实现条款1-11中所述的方法中的任何一种方法的步骤。

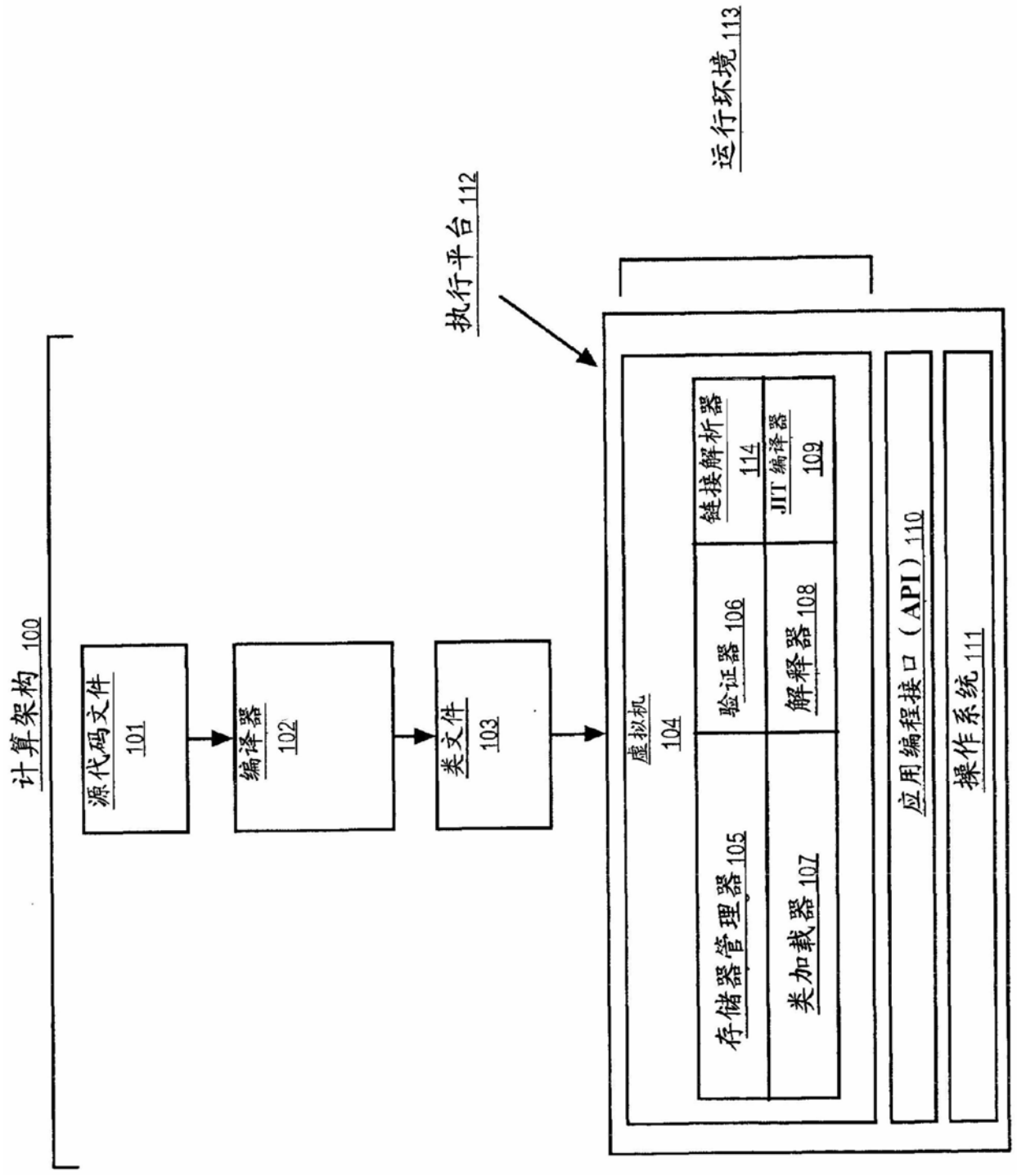

计算架构 100
源代码文件 101
编译器 102
类文件 103
执行平台 112
运行环境 113
虚拟机 104
存储器管理器 105
验证器 106
解释器 108
链接解析器 114
JIT 编译器 109
类加载器 107
应用编程接口（API）110
操作系统 111

图1

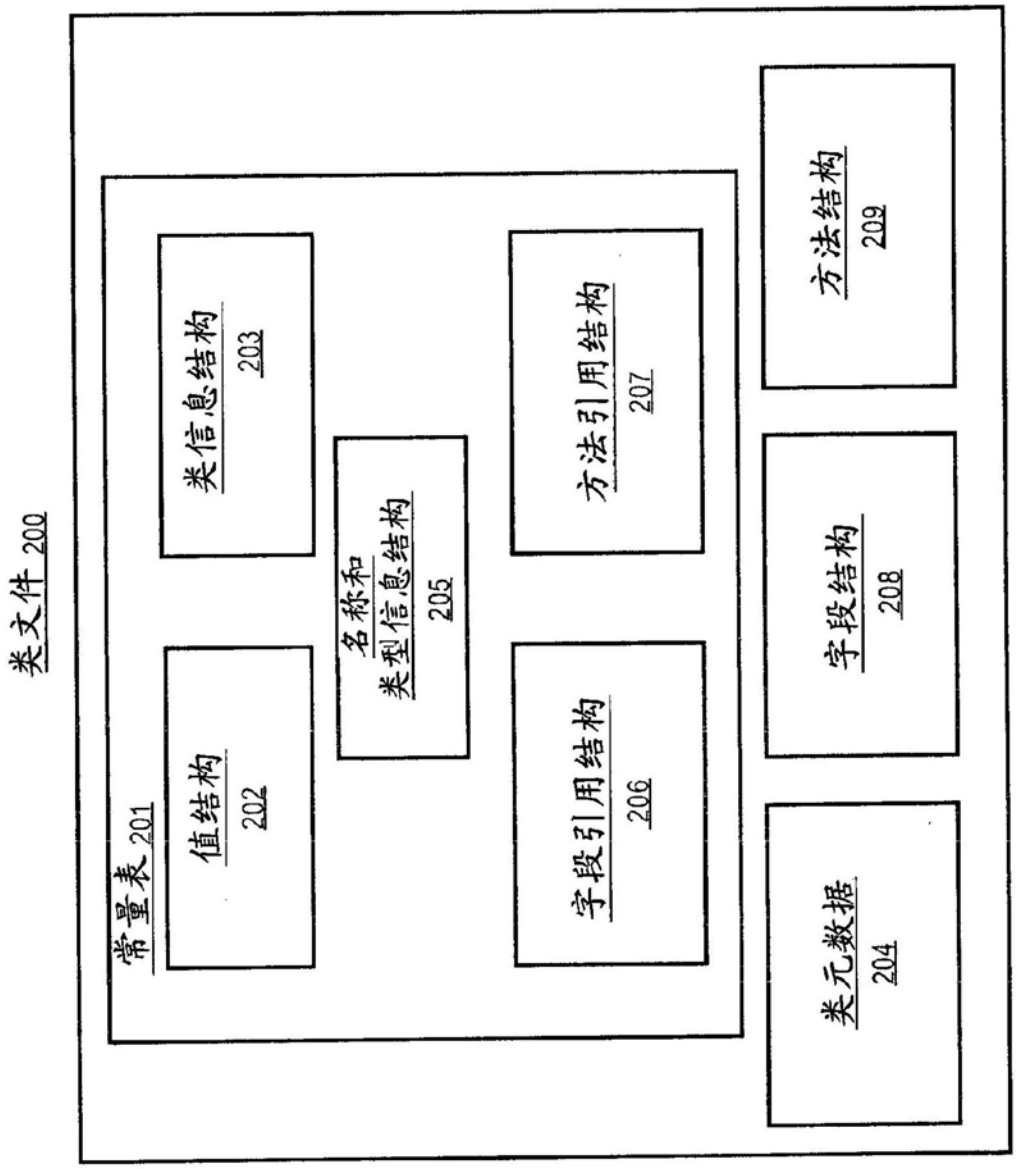

类文件 200
常量表 201
值结构 202
类信息结构 203
名称和类型信息结构 205
字段引用结构 206
方法引用结构 207
类元数据 204
字段结构 208
方法结构 209

图2

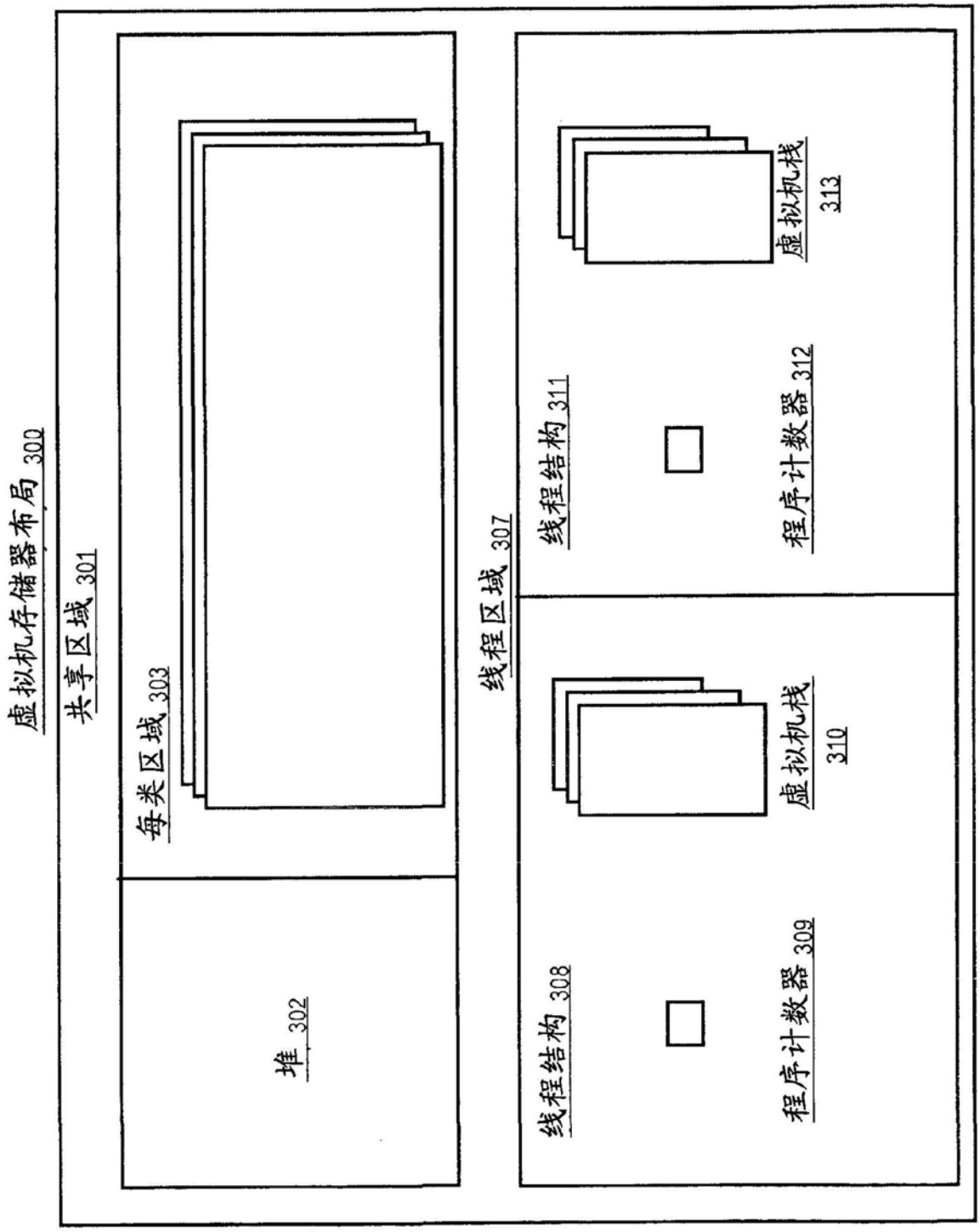

虚拟机存储器布局 300
共享区域 301
每类区域 303
堆 302
线程区域 307
线程结构 308
程序计数器 309
虚拟机栈 310
线程结构 311
程序计数器 312
虚拟机栈 313

图3

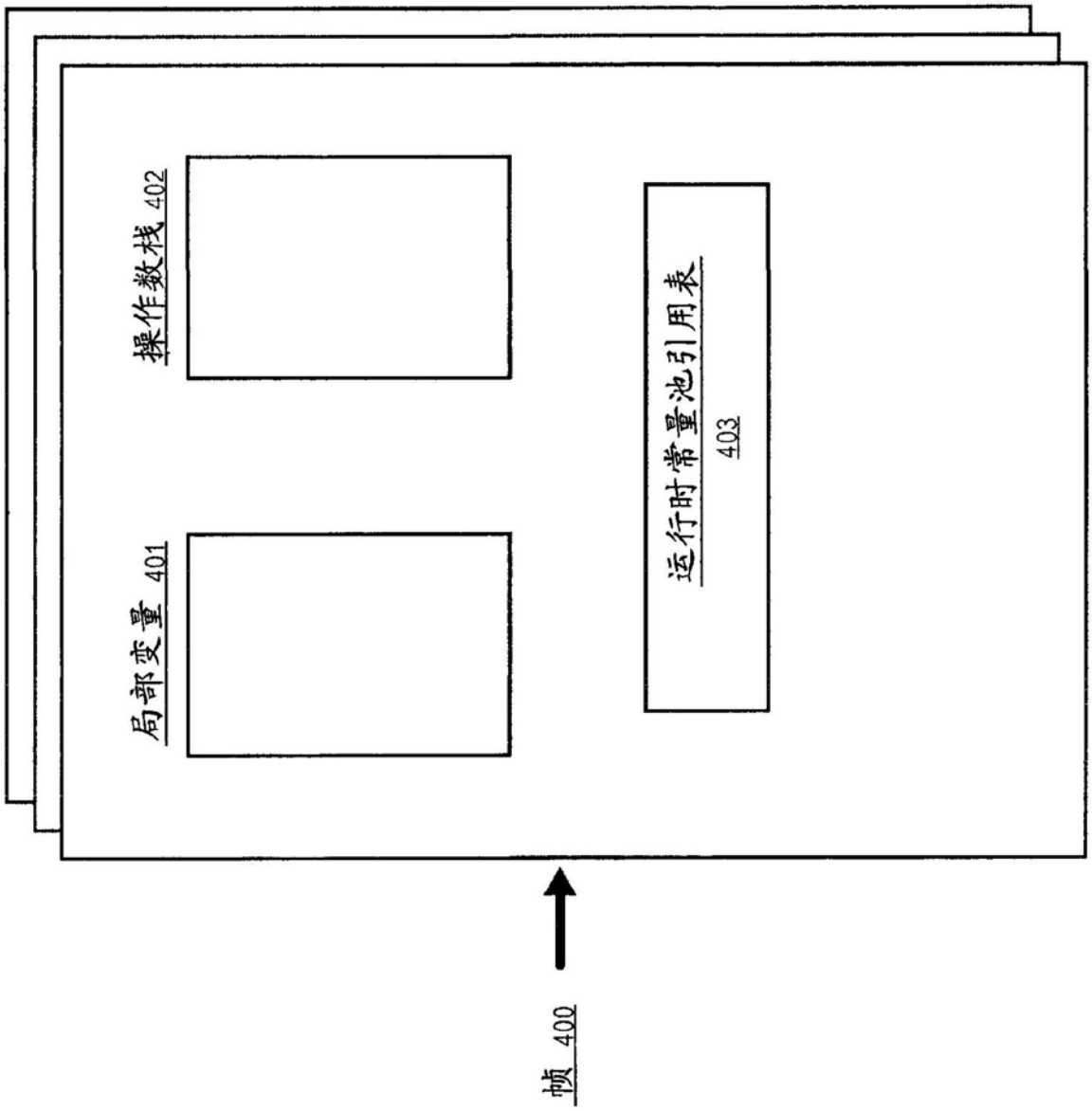


图4

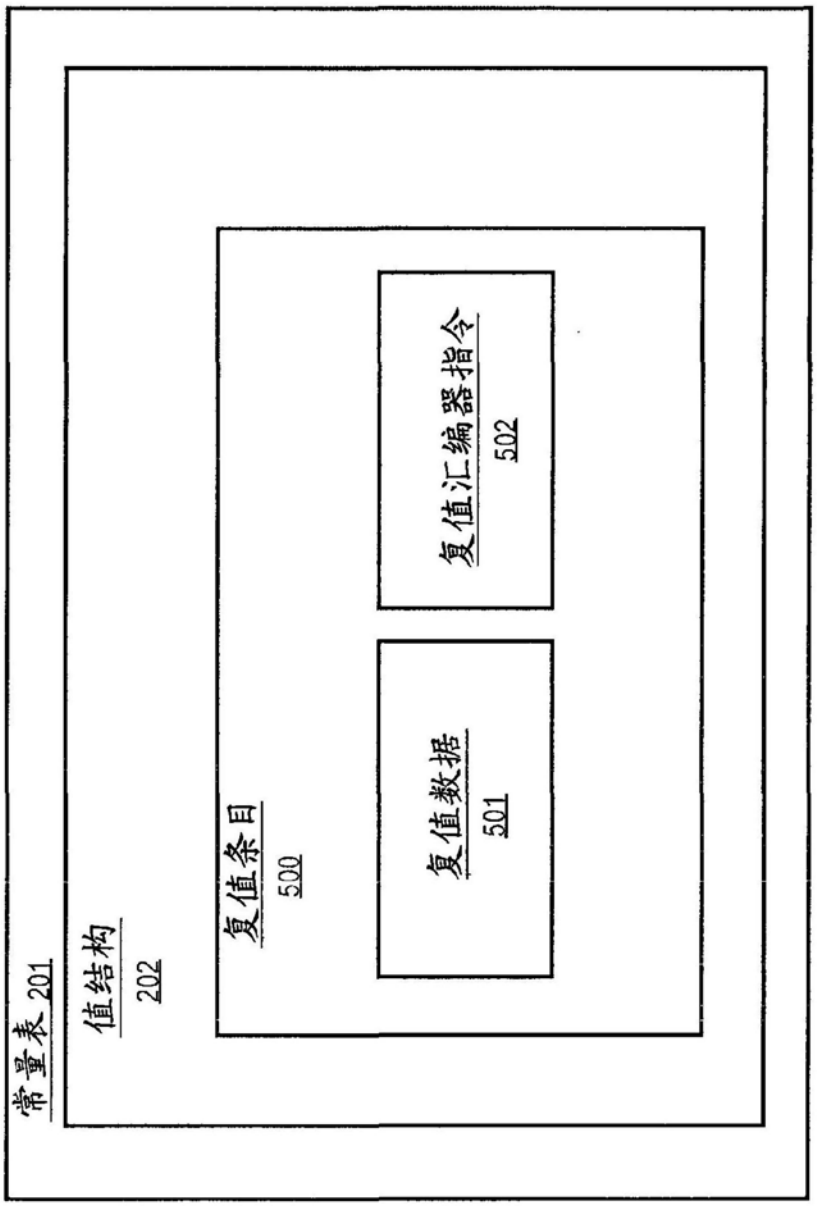


图5

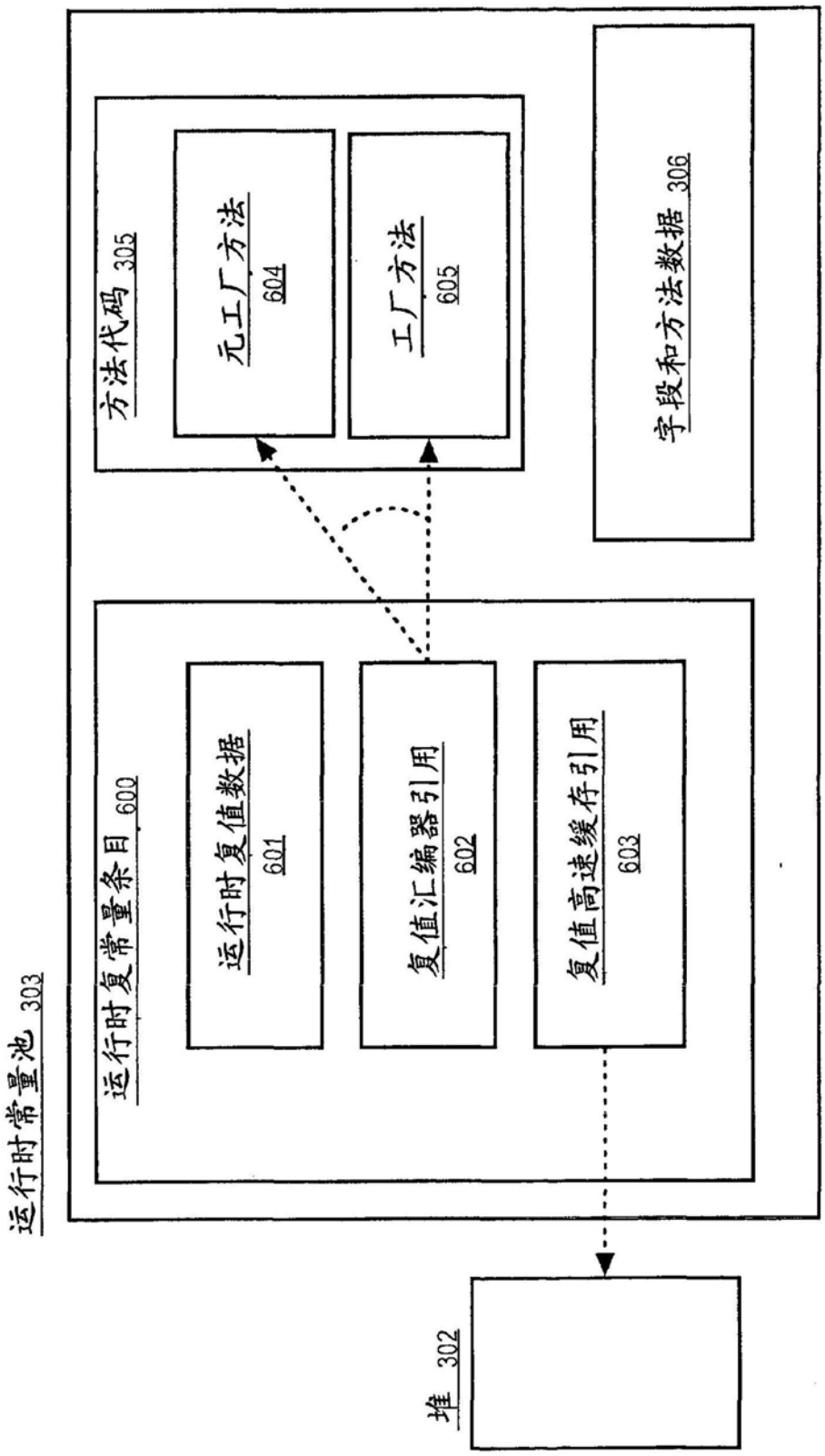

运行时常量池 303
运行时复常量条目 600
运行时复值数据 601
复值汇编器引用 602
复值高速缓存引用 603
方法代码 305
元工厂方法 604
工厂方法 605
字段和方法数据 306
堆 302

图6

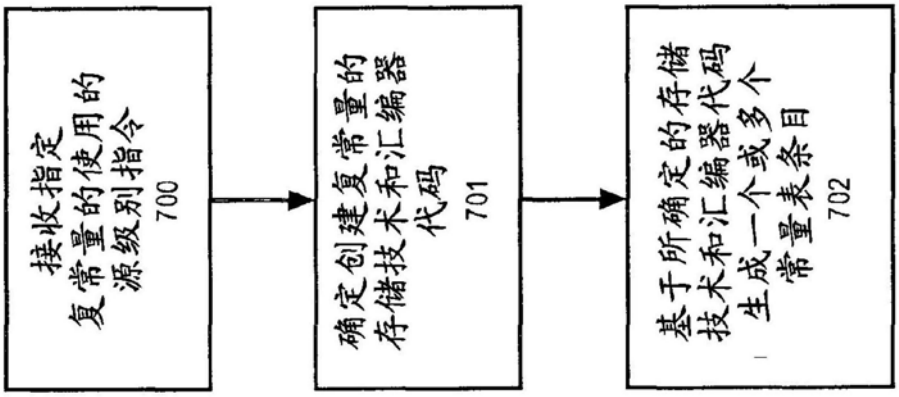

接收指定
复常量的使用的
源级别指令
700
确定创建复常量的
存储技术和汇编器
代码
701
基于所确定的存储
技术和汇编器代码
生成一个或多个
常量表条目
702

图7

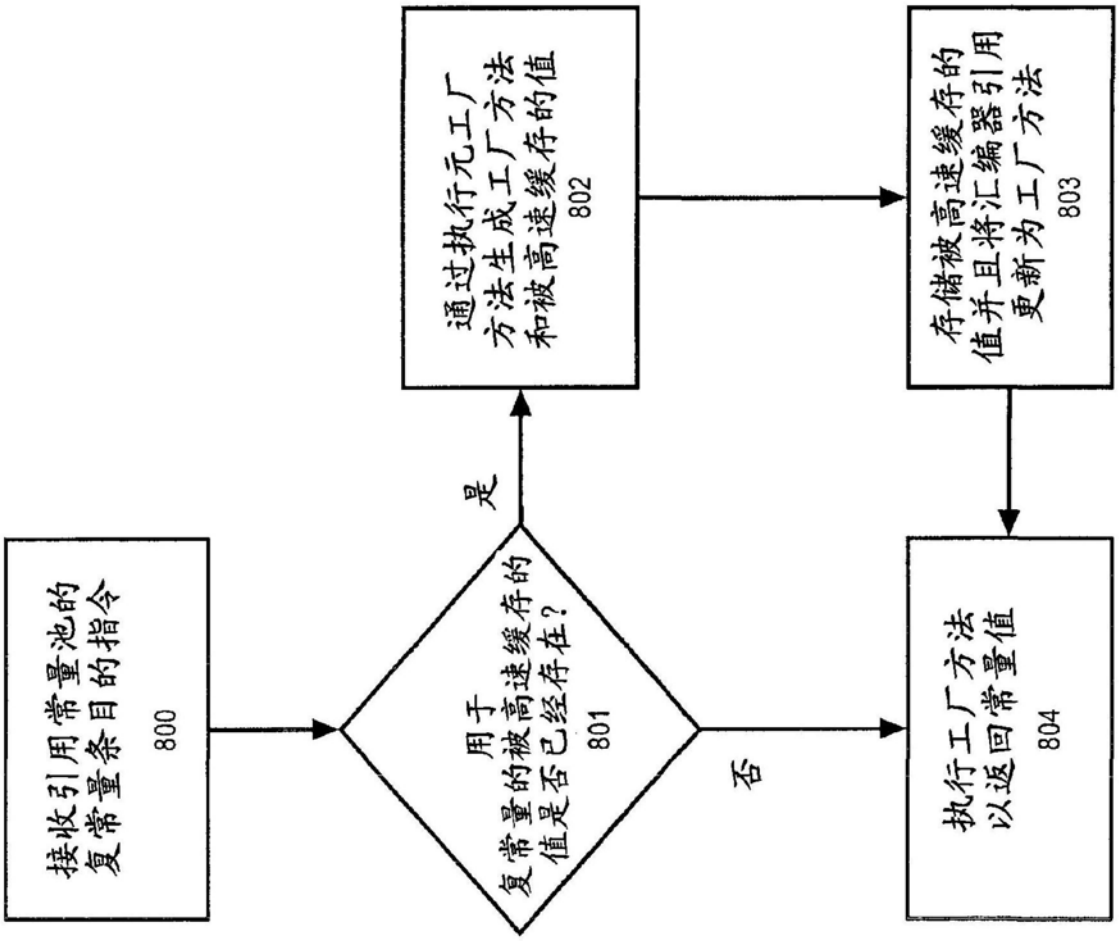

接收引用常量池的
复常量条目的指令
800
用于
复常量的被高速缓存的
值是否已经存在?
801
是
否
通过执行元工厂
方法生成工厂方法
和被高速缓存的值
802
存储被高速缓存的
值并且将汇编器引用
更新为工厂方法
803
执行工厂方法
以返回常量值
804

图8

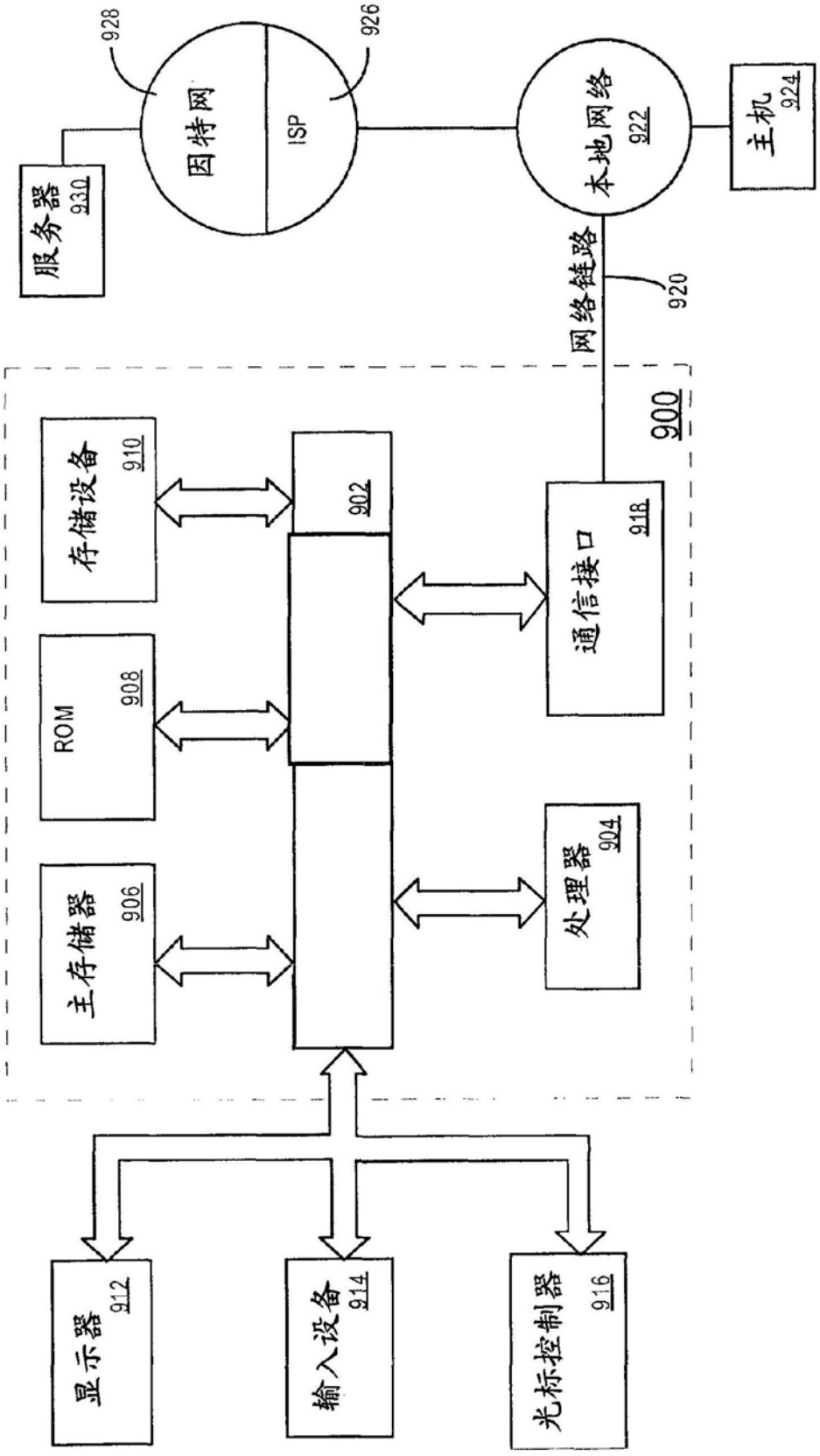

928
因特网
ISP
926
本地网络
922
主机
924
服务器
930
网络链路
920
900
存储设备
910
902
通信接口
918
ROM
908
主存储器
906
处理器
904
显示器
912
输入设备
914
光标控制器
916

图9