[19] 中华人民共和国国家知识产权局

[51] Int. Cl.
G06F 9/45 (2006.01)

[12] 发明专利申请公开说明书

[21] 申请号 200610008256.2

[43] 公开日 2006 年 10 月 18 日

[11] 公开号 CN 1848088A

[22] 申请日 2006.2.16
[21] 申请号 200610008256.2
[30] 优先权
[32] 2005. 2.18 [33] US [31] 11/062,306
[71] 申请人 国际商业机器公司
地址 美国纽约
[72] 发明人 A·H·基尔斯特拉
L·S·斯特帕尼安
K·A·斯图德雷

[74] 专利代理机构 北京市中咨律师事务所
代理人 于 静 李 峥

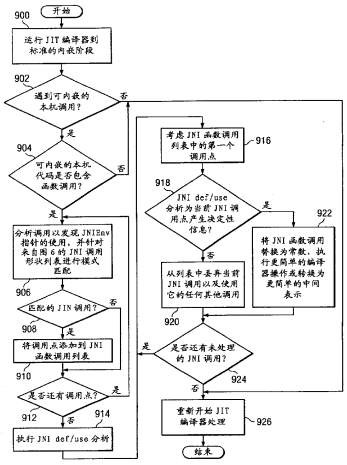
权利要求书 5 页 说明书 18 页 附图 4 页

[54] 发明名称

将本机接口函数调用转换为更简单操作的方法和系统

[57] 摘要

提供了用于将 Java 本机接口函数调用转换为常数、内部即时编译器操作或更简单的中间表示的方法和系统。 编译器生成用于多个本机接口函数调用的多个中间表示。 在内嵌本机代码期间，对每个本机函数调用执行匹配(针对该列表)，并生成本机接口函数调用的列表。 对于每个本机接口函数调用，JIT 调用转换器尝试根据该本机接口函数调用的类型将该本机接口函数调用替换为常数、内部即时编译器操作或更简单的中间表示。



1. 一种在数据处理系统中用于在即时编译期间将本机接口函数调用转换为更简单的操作的方法，该方法包括：

编译本机语言程序以生成用于多个本机接口函数调用的形状的列表；

通过将在该本机语言程序中进行的多个本机接口函数调用匹配于该形状列表，来识别用于可能的转换的本机接口函数调用的列表；

通过跟踪作为参数被传递给本机接口函数的多个变元的多个值，来对所识别的本机接口函数调用的列表中的每一个执行定义和使用分析；以及

使用所述定义和使用分析的结果，将所述本机接口函数调用的列表的一部分转换为常数、内部即时编译器操作、和更简单的中间表示中的一个。

2. 根据权利要求 1 的方法，其中所述形状列表包括所述多个本机接口函数调用的中间表示，并包括该多个本机函数调用对本机接口环境变量和用户变元的用法。

3. 根据权利要求 2 的方法，其中所述识别步骤是在内嵌本机函数调用期间、响应于检测到内嵌的本机函数调用中的本机接口函数调用而执行的，且其中所述识别步骤还包括：

对于所述识别的本机接口函数调用列表中的每一个，确定所述本机接口环境变量是否是以在所述生成的形状的列表中的一形状中的相同方式被使用的；

如果所述本机接口环境变量是以在所述生成的形状的列表中的一形状中的相同方式被使用的，则确定在该形状和该本机接口函数调用的中间表示之间是否存在匹配，其中如果该本机接口函数调用的中间表示与该形状相比具有相同数量的变元，且该本机

接口函数调用中的每个变元与该形状中的变元的类型一致，则存在匹配；以及

如果在该形状和该本机接口函数调用的中间表示之间存在匹配，则将该本机接口函数调用添加到所述本机接口函数调用列表，其中该本机接口函数调用列表是按照在所述多个本机接口函数调用的中间表示中出现的顺序排序的。

4. 根据权利要求 3 的方法，其中所述定义和使用分析包括在识别了所述本机接口函数调用列表之后跟踪被传递给该本机接口函数调用列表的多个对象到该多个对象作为变元被传递给本机接口函数的位置。

5. 根据权利要求 4 的方法，其中所述转换步骤包括：

对于所述本机接口函数调用列表中的每个本机接口函数调用，确定所述定义和使用分析是否返回决定性的结果；

如果该定义和使用分析返回决定性的结果，则确定该本机接口函数调用的类型；以及

根据该本机接口函数调用的类型，将该本机接口函数调用替换为常数值、内部即时编译器操作、和更简单的中间表示中的一个。

6. 根据权利要求 5 的方法，还包括：

如果所述定义和使用分析返回已知的结果，则将所述调用点替换为包含条件逻辑的、代表所指定的调用的语义的中间表示；

如果该定义和使用分析返回未知的结果，则从所考虑的本机接口调用列表中丢弃该本机接口函数调用和使用该本机接口函数调用的任何其他本机接口函数调用。

7. 根据权利要求 5 的方法，其中所述常数、内部即时编译器操作和更简单的中间表示提供了对虚拟机的服务和数据的直接的和更快的访问，其中所述内部即时编译器操作执行其功能而不使用本机接口。

8. 根据权利要求 1 的方法，其中所述形状列表中的形状包括以中间表示来表示的用于访问实际目标本机接口函数的手段和以中间表示来表示的每个实际变元接收到的处理，且其中该形状唯一地标识了该实际目标本机接口函数。

9. 根据权利要求 1 的方法，其中所述定义和使用分析确定定义了什么变量以及在何处使用所定义的变量。

10. 一种用于在即时编译期间将本机接口函数调用转换为更简单的操作的数据处理系统，该数据处理系统包括：

即时编译器，用于编译本机语言程序以生成用于该本机语言程序中的多个本机接口函数调用的形状的列表；

即时编译器内嵌器，用于从该形状列表中识别用于可能的转换的本机接口函数调用的列表；

其中该即时编译器对所述本机接口函数调用的列表和被传递给所述本机接口函数的多个变元执行定义和使用分析，其中该定义和使用分析确定定义了什么变量，以及在何处使用所定义的变量；以及

即时调用转换器，用于使用该定义和使用分析的结果，将所述本机接口函数调用的列表的一部分转换为常数、内部即时编译器操作、和更简单的中间表示中的一个。

11. 根据权利要求 10 的数据处理系统，其中所述形状列表包括所述多个本机接口函数调用的中间表示，并包括该多个本机函数调用对本机接口环境变量和用户变元的用法。

12. 根据权利要求 11 的数据处理系统，其中所述即时编译器内嵌器在内嵌本机函数调用期间、响应于检测到内嵌的本机函数调用中的本机接口函数调用而识别用于转换的所述本机接口函数调用列表，且其中该即时编译器内嵌器还执行以下操作：

对于所述识别的本机接口函数调用列表中的每一个，确定所述本机接口环境变量是否是以在所述生成的形状的列表中的一形

状中的相同方式被使用的；

如果所述本机接口环境变量是以在所述生成的形状的列表中的一形状中的相同方式被使用的，则确定在该形状和该本机接口函数调用的中间表示之间是否存在匹配，其中如果该本机接口函数调用的中间表示与该形状相比具有相同数量的变元，且该本机接口函数调用中的每个变元与该形状中的变元的类型一致，则存在匹配；以及

如果在该形状和该本机接口函数调用的中间表示之间存在匹配，则将该本机接口函数调用添加到所述本机接口函数调用列表，其中该本机接口函数调用列表是按照在所述多个本机接口函数调用的中间表示中出现的顺序排序的。

13. 根据权利要求 12 的数据处理系统，其中所述即时编译器内嵌器在识别了所述本机接口函数调用列表之后跟踪被传递给该本机接口函数调用列表的多个对象到该多个对象作为变元被传递给本机接口函数的位置。

14. 根据权利要求 13 的数据处理系统，其中所述即时调用转换器还执行以下操作：

对于所述本机接口函数调用列表中的每个本机接口函数调用，确定所述定义和使用分析是否返回已知的结果；

如果该定义和使用分析返回已知的结果，则确定该本机接口函数调用的类型；以及

根据该本机接口函数调用的类型，将该本机接口函数调用替换为常数值、内部即时编译器操作、和更简单的中间表示中的一个。

15. 根据权利要求 14 的数据处理系统，其中所述即时调用转换器还执行以下操作：

如果该定义和使用分析返回未知的结果，则从所述本机接口调用列表中丢弃该本机接口函数调用和使用该本机接口函数调用

的任何其他本机接口函数调用。

16. 根据权利要求 14 的数据处理系统，其中所述内部即时编译器操作提供了对虚拟机的服务和数据的直接的和更快的访问，其中所述内部即时编译器操作执行其功能而不使用本机接口。

17. 一种在计算机可读存储介质中的用于在即时编译期间将本机接口函数调用转换为更简单的操作的计算机程序产品，该计算机程序产品包括：

用于执行前述方法权利要求的任何方法的指令。

将本机接口函数调用转换为更简单操作的方法和系统

技术领域

本发明涉及改进的数据处理系统。具体地，本发明涉及数据处理系统中的 Java 本机函数调用。更具体地，本发明涉及在数据处理系统中将 Java 本机接口函数调用转换为常数、内部编译器操作或更简单的操作。

背景技术

Java 是一种面向对象的编程语言和环境，其专注于将数据定义为对象以及可应用于这些对象的方法。Java 仅支持单继承，这意味着每个类在任何给定时间仅能从一个其他类继承。Java 也允许创建被称为接口的完全抽象的类，这种类允许定义可与若干类共享的方法，而不管其他类如何处理这些方法。Java 提供了分发软件和扩展 Web 浏览器的能力的机制，因为程序员能编写小应用程序一次，而该小应用程序能在 Web 上的任何 Java 使能的机器上运行。

Java 虚拟机（JVM）是一种虚拟计算机组件。JVM 允许 Java 程序在不同平台上被执行，而不是仅在为其编译代码的一个平台上被执行。Java 程序是为 JVM 编译的。以这种方式，Java 能够支持用于很多类型的数据处理系统的应用，这些数据处理系统可包含多种中央处理单元和操作系统体系结构。为了使得 Java 应用能在不同类型的数据处理系统上执行，编译器通常生成一体系结构中性的文件格式—编译的代码可在很多处理器上执行，倘若存在 Java 运行时系统的话。Java 编译器生成非特定于具体计算机体系结构的字节码指令。字节码是由 Java 编译器生成并由 Java 解释器执行的独立于机器的代码。Java 解释器是 JVM 中的交替地解码和执行一个或多个字节码的模块。这些字节码指令被设计为易于在任何机器上解释，

并易于被动态地（on the fly）转换为本机机器码。

可使用开发环境，例如可从 Sun Microsystems 公司获得的 Java 开发工具包（JDK），来从 Java 语言源代码和库构建 Java 字节码。可将这种 Java 字节码存储为 Web 服务器上的 Java 应用或小应用程序，可通过网络将该 Java 应用或小应用程序从该 Web 服务器下载到用户的机器并在本地 JVM 上执行。

Java 运行时环境被特别设计为限制 Java 应用可能对它运行于其上的系统造成的损害。这对于万维网尤其重要，在万维网上当用户访问包含 Java 小应用程序的网页时，Java 小应用程序被自动下载和执行。一般说来，人们不想执行任意的程序；它们可能包含病毒，或者它们甚至可能本身就是潜在恶意的，而不仅是无意地携带了不受欢迎的代码。除非用户特别地允许它（通过设置到 JVM 的用户接口中的适当标志），Java 小应用程序不能向附加的存储设备（除非可能向一特定的、受限制的区域）读或写，它也不能向存储器位置（除非向一特定的、受限制的区域）读或写。

不仅 Java 小应用程序是为通过网络下载设计的，标准的 Java 库也特别地支持客户机－服务器计算。Java 语言包括用于多线程和用于网络通信的规定。与其他语言（例如 C）相比，编写这样的一对程序更容易得多，一个程序在用户的计算机上本地执行并处理用户交互，另一个在服务器上远程执行并可能执行更复杂和处理器密集的工作。

尽管 Java 语言被设计为是独立于平台的并主要在安全环境中执行，程序员可通过 Java 本机接口（JNI）使用 C 风格的调用约定来使用主机操作系统上的编译的本机二进制代码，从而扩展 Java 应用。以这种方式，Java 应用可具有对主机操作系统的完全访问权，包括对附加的 I/O 设备、存储器等的读或写。由于此，Java 程序可以成为特定于平台的代价完成通过 JVM 通常不允许的任务。然而，使用设计良好的体系结构，Java 语言程序员可干净地将独立于平台的部分隔离，并向其他 Java 组件呈现一干净的、独立于平台的对象 API，而同时完成特定于平台的任务。

使用 JNI API，位于 Java 虚拟机（JVM）内的参数和类数据以及由

JVM 提供的服务可被访问并可被修改。然而，由于 JNI 暴露了一非常独立于平台和不透明的 JVM 的表示、其数据和服务，所以存在着开销。JNI API 通常是由一指向函数指针（这些函数指针指向 JNI 函数实现）的表的指针（被称为 JNI 环境指针，或 JNIEnv）暴露的。虽然该指针对于 JNI 的平台独立性是必需的，该指针也可能是代价高昂的，因为在 JNI 函数调用期间执行额外的分支和表查找。JNI 调用的代价取决于被调用的 JNI 函数的类型。

例如，为了访问从 Java 传递给本机代码的串和数组参数，需要对 JNI 函数的特殊调用。这些特殊的调用造成昂贵的运行时复制操作以避免触及 JVM 的数据副本。因此 JNI API 提供了访问器（accessor）函数，这些函数试图增加本机代码接收对底层 JVM 数据的直接引用的机会。然而，这些函数是根据 JVM 的决定实现的，并且这些函数的使用对程序员的自由施加了某些限制。

对于字段和方法访问，也要求 JNI 函数调用来从本机代码修改对象和访问 JVM 服务。例如，为了修改对象的字段或调用类的方法，必须首先检索适当数据的句柄。这种检索通常被实现为在 JVM 的反映（reflective）数据结构上的遍历以及在运行时昂贵的基于串的签名比较操作。这种遍历比 Java 中的直接字段访问慢几个数量级。此外，如果非异步的 JVM 正在执行阻塞（blocking）工作，则 JNI 函数调用可能停滞（stall）。

其他 JNI 函数具有类似于用于字段和方法访问的 JNI 函数的唯一的一组开销。这些 JNI 函数包括实例化对象、管理引用、处理异常、支持同步和反映的函数，以及在 Java 调用 API 中用于将 JVM 嵌入本机代码内的函数。

已做出了若干尝试来最小化 JNI 调用的开销。最初的尝试是基于程序员的优化，其形式为当编写使用 JNI API 的本机代码时高效的编码技术。JNI 规范也提供了一组可返回对 JVM 数据和对象的直接引用的关键函数，并建议了避免调用 JNI API 的方法，例如在类的静态初始化期间缓存字段和方法 ID。

另一种尝试涉及通过限制在本机代码中可提供的功能的类型来最小化对 JNI API 函数的依赖，从而消除与本机函数相关的开销。然而，这种机制只能用于被保证不会要求垃圾收集、异常处理、同步或任何类型的安全支持的方法，因为这些是包装器（wrapper）提供的功能类型。

在又一种最化小 JNI 函数调用的开销的尝试中，编译器可使用紧密耦合于 VM 的专有本机接口，因为该接口具有关于 VM 的内部结构的知识。这种编译器的一个例子是 Jcc 编译器，这是一种使用专有本机接口直接编译到本机代码的优化编译器，该专有本机接口令人联想到来自 Sun Microsystems 公司的原始的本机方法接口（NMI）。然而，该 NMI 已被 JNI 代替。在 Jcc 中，被标记为“本机的”Java 方法将被当作这样的 C 方法，该 C 方法将使用 C 调用约定被调用，并对每个 Java 类发出一 C 结构。以这种方式，Jcc 可内嵌短的汇编段以加速本机调用。

尽管上述各尝试最小化了与 JNI 函数调用相关的某些开销，但并不存在不限制在本机代码中实现的功能的类型的（如在上述第二种尝试中提及的）、或不耦合到特定 VM 实现的（如在上述第三种尝试中提及的）现有的机制。

因此，拥有这样的方法、装置、和计算机指令将是有利的，该方法、装置和计算机指令由即时（JIT）编译器使用以将 JNI 函数调用转换为常数、内部编译器操作或更简单的中间表示，从而可改进本机代码访问 JVM 数据和服务的性能，而不耦合到特定的 VM 实现或牺牲 JNI 的类型安全（type safety）。

发明内容

本发明提供了用于在即时编译期间将本机函数调用转换为更简单的操作的方法、装置、和计算机指令。在编译一程序以生成用于多个 Java 本机接口（JNI）函数调用的形状（shape）的列表后，提供一内嵌器以识别用于转换的、在本机代码中进行的本机接口函数或 JNI 调用的列表。

然后，即时（JIT）编译器对该本机接口函数调用列表和被传递给这些

本机接口函数调用的多个变元执行定义和使用（def/use）分析。随后，提供一 JIT 调用转换器以使用所述定义和使用分析的结果将该本机接口函数调用列表的一部分转换为常数、内部即时编译器操作或更简单的中间表示。

在本机函数调用的内嵌期间，即时内嵌器对于在内嵌的代码中发生的每个本机接口调用确定在该调用中使用的本机接口环境变量是否位于与它在代表所有已知的本机接口调用的预定义形状的列表中所出现的相同的位置。一 JNI 调用的形状包括（1）访问实际目标本机接口或 JNI 函数的手段（以中间表示表示的）（2）每个实际变元接收到的处理（同样以中间表示表示的）。典型的 JNI 调用的形状（当构建形状列表时被创建的）唯一地确定实际 JNI 例程。如果在调用中使用的本机接口环境变量位于与在代表本机接口调用的预定义形状的列表中的形状之一中相同的位置，则内嵌器确定在该形状和由一内嵌的本机函数进行的 JNI 调用的中间表示之间是否存在匹配。

如果在该形状和该 JNI 函数调用的中间表示之间存在匹配，则内嵌器将该 JNI 调用添加到用于当前被内嵌的本机方法的 JNI 函数调用的列表中，其中该调用列表是以在该多个 JNI 调用的中间表示中出现的顺序排序的。

在执行了定义和使用分析之后，该即时调用转换器对于该本机接口函数调用列表中的每个本机接口函数调用确定本机接口函数调用的类型，并尝试根据每个 JNI 函数调用的类型将该本机接口函数调用替换为常数值、内部即时编译器操作、或更简单的中间表示。

因为新生成的常数、内部即时编译器操作、或更简单的中间表示不使用 JNI 而执行，它们转而提供了对 Java 虚拟机的服务和数据的直接的和更快的访问。

附图说明

在所附权利要求中提出了相信为本发明的特点的新颖特征。然而，通过结合附图参照以下对示例性实施例的详细说明可最好地理解本发明本身

以及其优选使用方式、其他目标和优点，在这些附图中：

图 1 是根据本发明的一优选实施例可在其中实现本发明的数据处理系统的图示；

图 2 是可在其中实现本发明的数据处理系统的框图；

图 3 是示出了在可实现本发明的计算机系统内运行的软件组件的关系的框图；

图 4 是根据本发明的一优选实施例的 JVM 的框图；

图 5 示出了根据本发明的一优选实施例由 JNI 调用转换器执行的 JNI 函数调用转换；

图 6 示出了根据本发明的一优选实施例为 JNI 函数调用生成中间表示；

图 7 示出了根据本发明的一优选实施例用于 JNI 转换的本机内嵌准备；

图 8 的图示出了根据本发明的一优选实施例由编译器执行的示例性 def/use 分析；以及

图 9 是根据本发明的一优选实施例用于将 JNI 函数调用转换为常数、内部编译器操作或更简单的中间表示的示例性过程。

具体实施方式

现参照附图并具体参照图 1，其示出了根据本发明的一优选实施例可在其中实现本发明的数据处理系统的图示。示出了一计算机 100，其包括系统单元 102、视频显示终端 104、键盘 106、可包括软盘驱动器和其他类型的永久的和可拆装的存储介质的存储设备 108、和鼠标 110。个人计算机 100 中还可包括其他输入设备，例如游戏杆、触控板、触屏、跟踪球、麦克风等。可使用任何适当的计算机，例如位于纽约 Armonk 的国际商业机器公司的产品 IBM eServer 计算机或 IntelliStation 计算机，来实现计算机 100。虽然所示的图示示出了一计算机，本发明的其他实施例可在其他类型的数据处理系统例如网络计算机中实现。计算机 100 还优选地包括图形用

户界面（GUI），该图形用户界面可通过存在于在计算机 100 中运行的计算机可读介质中的系统软件来实现。

现参照图 2，其示出了可在其中实现本发明的数据处理系统的框图。数据处理系统 200 是一计算机例如图 1 中的计算机 100 的示例，实现本发明的过程的代码或指令可位于该计算机中。数据处理系统 200 使用外围部件互连（PCI）局部总线体系结构。尽管所示的示例使用 PCI 总线，也可使用诸如加速图形端口（AGP）和工业标准结构（ISA）的其他总线体系结构。处理器 202 和主存储器 204 通过 PCI 桥 208 连接到 PCI 局部总线 206。PCI 桥 208 也可包括集成的存储控制器和用于处理器 202 的高速缓冲存储器。可通过直接部件互连或通过附加连接器进行附加的到 PCI 局部总线的连接。在所示的示例中，局域网（LAN）适配器 210、小型计算机系统接口（SCSI）主机总线适配器 212、和扩展总线接口 214 通过直接部件互连连接到 PCI 局部总线 206。相反地，音频适配器 216、图形适配器 218、和音频/视频适配器 219 通过插入到扩展槽中的附加板连接到 PCI 局部总线 206。扩展总线接口 214 提供了用于键盘和鼠标适配器 220、调制解调器 222、和附加存储器 224 的连接。SCSI 主机总线适配器 212 提供了用于硬盘驱动器 226、磁带驱动器 228、和 CD-ROM 驱动器 230 的连接。典型的 PCI 局部总线实现将支持三个或四个 PCI 扩展槽或附加连接器。

操作系统运行在处理器 202 上并用于对图 2 中的数据处理系统 200 中的各部件进行协调和提供控制。操作系统可以是商业上可获得的操作系统例如可从 Microsoft 公司获得的 Windows XP。面向对象的编程系统例如 Java 可与操作系统一起运行，并提供从在数据处理系统 200 上运行的 Java 程序或应用到操作系统的调用。“Java”是 Sun Microsystems 公司的商标。用于操作系统、面向对象的编程系统、和应用或程序的指令位于存储设备例如硬盘驱动器 226 上，并可被加载到主存储器 204 中以便由处理器 202 执行。

本领域的普通技术人员将理解图 2 中的硬件可根据实现而变化。作为对图 2 中所示的硬件的附加或替代，可使用其他内部硬件或外围设备例如

快闪只读存储器（ROM）、等效的非易失性存储器、或光盘驱动器等。此外，可将本发明的过程应用于多处理器数据处理系统。

例如，数据处理系统200如果可选地被配置为网络计算机，则可不包括SCSI主机总线适配器212、硬盘驱动器226、磁带驱动器228、和CD-ROM 230。在这种情况下，将被适当地称为客户端计算机的该计算机包括某种类型的网络通信接口，例如LAN适配器210、调制解调器222等。作为另一个示例，数据处理系统200可以是独立的系统，其被配置为不依赖于某种类型的网络通信接口而可引导，不管数据处理系统200是否包括某种类型的网络通信接口。作为进一步的示例，数据处理系统200可以是个人数据助理（PDA），其被配置为具有ROM和/或快闪ROM，以为存储操作系统文件和/或用户生成的数据提供非易失性存储器。

图2中所示的示例和以上描述的示例并非意在暗示结构上的限制。例如，数据处理系统200除了采取PDA的形式也可以是笔记本计算机或手持式计算机。数据处理系统200也可以是信息站（kiosk）或网络电器（web appliance）。

本发明的过程是由处理器202使用计算机实现的指令执行的，这些指令可被加载到存储器例如主存储器204、存储器224、或一个或多个外围设备226－230中。

现参照图3，该框图示出了在可实现本发明的计算机系统内运行的各软件组件的关系。基于Java的系统300包含特定于平台的操作系统302，该操作系统为在特定硬件平台上执行的软件提供了硬件和系统支持。JVM 304是一个可与操作系统一起执行的软件应用。JVM 304提供了Java运行时环境，该环境具有执行Java应用或小应用程序306的能力，所述Java应用或小应用程序是以Java编程语言编写的程序、小服务程序、或软件组件。JVM 304在其中运行的计算机系统可以类似于以上描述的数据处理系统200或计算机100。然而，JVM 304可在具有嵌入的picoJava内核的所谓的Java芯片、硅上Java（java-on-silicon）、或Java处理器上的专用的硬件中实现。

在Java运行时环境的中心的是JVM，该JVM支持Java的环境的所有方面，包括其体系结构、安全特征、跨网络的移动性、和平台独立性。

JVM是一虚拟计算机，即被抽象地规定的计算机。该规范定义了每个JVM必须实现的某些特征，并具有可取决于JVM被设计为在其上执行的平台的一定范围的设计选择。例如，所有JVM必须执行Java字节码，并可使用一系列技术来执行由字节码代表的指令。JVM可完全在软件中实现或在某种程度上在硬件中实现。这种灵活性允许为大型计算机和PDA设计不同的JVM。

JVM是实际执行Java程序的虚拟计算机组件的名称。Java程序不是直接由中央处理器运行的，而是由JVM运行的，JVM本身是运行在处理器上的一个软件。JVM允许Java程序在不同的平台上执行，而不是仅在为其编译代码的一个平台上执行。Java程序是为JVM编译的。以这种方式，Java能够支持用于很多类型的数据处理系统的应用，这些数据处理系统可包含各种中央处理单元和操作系统体系结构。为了使Java应用能在不同类型的数据处理系统上执行，编译器典型地生成体系结构中性的文件格式—编译的代码可在很多处理器上执行，只要存在Java运行时系统即可。Java编译器生成非特定于具体计算机体系结构的字节码指令。字节码是由Java编译器生成并由Java解释器执行的独立于机器的代码。Java解释器是交替地解码和解释一个或多个字节码的、JVM的部分。这些字节码指令被设计为易于在任何计算机上执行并易于被动态地（on the fly）转换为本机机器码。字节码可由即时编译器或JIT转换为本机代码。

JVM加载类文件并执行其中的字节码。类文件是由JVM中的类加载器加载的。类加载器从应用加载类文件并从应用所需要Java应用编程接口（API）加载类文件。执行字节码的执行引擎可随不同的平台和实现而不同。

一种类型的基于软件的执行引擎是JIT编译器。使用这种类型的执行，在成功地满足用于JIT编译一方法的标准后，将一方法的字节码编译为本机机器码。然后用于该方法的本机机器码被缓存，并在下次调用该方法时

被重用。执行引擎也可在硬件中实现并嵌入芯片中，从而字节码被本机地执行。JVM 通常解释字节码，但 JVM 也可使用其他技术，例如即时编译，来执行字节码。

当在特定于平台的操作系统上的软件中实现的 JVM 上执行应用时，Java 应用可通过调用本机方法来与主机操作系统交互。Java 方法是在 Java 语言中编写的，被编译为字节码，并被存储在类文件中。本机方法是在某种其他语言中编写的，并被编译为特定处理器的本机机器码。本机方法被存储在动态链接库中，动态链接库的精确形式是特定于平台的。

现参照图 4，其示出了根据本发明的一优选实施例的 JVM 的框图。JVM 404 包括类加载器子系统 402，该子系统是一用于在给定了类型的完全限定名的情况下加载类型例如类和接口的机制。JVM 404 还包含运行时数据区 404、执行引擎 406、本机方法接口 408、和存储管理 410。执行引擎 406 是一用于执行包含在由类加载器子系统 402 加载的类的方法中的指令的机制。执行引擎 406 可以是例如 Java 解释器 412 或即时编译器 414。本机方法接口 408 允许访问底层操作系统中的资源。本机方法接口 408 可以是例如 Java 本机接口（JNI）。

运行时数据区 404 包含本机方法栈 416、Java 栈 418、PC 寄存器 420、方法区 422、和堆 424。这些不同的数据区代表 JVM 404 执行程序所需要的存储区的结构。

Java 栈 418 用于存储 Java 方法调用的状态。当发起新的线程时，JVM 为该线程创建一新的 Java 栈。JVM 只直接在 Java 栈上执行两种操作：它推入和弹出帧。一线程的 Java 栈存储用于该线程的 Java 方法调用的状态。Java 方法调用的状态包括其局部变量、调用它所用的参数、其返回值（如果有的话）、和中间计算。Java 栈由栈帧组成。一栈帧包含单个 Java 方法调用的状态。当一线程调用一方法时，JVM 将一新的栈帧推入该线程的 Java 栈。当该方法完成时，JVM 将用于该方法的帧弹出并丢弃它。JVM 没有任何用于保存中间值的寄存器；任何需要或产生中间值的 Java 指令使用该栈来保存中间值。以这种方式，为多种平台体系结构良好地定义了

Java 指令集。

程序计数器（PC）寄存器 420 用于指示下一个要执行的指令。每个实例化的线程获得它自己的 PC 寄存器和 Java 栈。如果该线程正在执行一 JVM 方法，则该 PC 寄存器的值指示下一个要执行的指令。本机方法栈 416 存储本机方法的调用的状态。本机方法调用的状态是以依赖于实现的方式存储在本机方法栈、寄存器、或其他依赖于实现的存储区中的。在某些 JVM 实现中，本机方法栈 416 和 Java 栈 418 是结合在一起的。

方法区 422 包含类数据，而堆 424 包含所有实例化的对象。在这些示例中，常数池位于方法区 422 中。JVM 规范严格地定义了数据类型和操作。大多数 JVM 选择具有一个方法区和一个堆，每一个都由运行在该 JVM 例如 JVM 404 中的所有线程共享。当 JVM 404 加载类文件时，它根据包含在该类文件中的二进制数据分析关于类型的信息。JVM 404 将这种类型信息放入方法区。每次创建一类实例或数组，就从堆 424 中分配用于该新对象的存储空间。JVM 404 包括一在用于堆 424 的存储区中分配存储空间的指令，但不包括用于在该存储区中释放该空间的指令。所示的示例中的存储管理 410 管理分配给堆 424 的存储区中的存储空间。存储管理 410 可包括一垃圾收集器，该垃圾收集器自动回收由不再被引用的对象所使用的存储空间。此外，垃圾收集器也可以移动对象以减少堆的碎片化。

本发明提供了在 JIT 编译期间用于将 JNI 函数调用转换为更简单的编译器操作（其最终导致生成常数和更高效的中间表示）的方法、装置、和计算机指令。在一优选实施例中，本发明利用从内嵌的本机函数中间表示转换来的 JIT 编译器中间表示。本发明提供了一 JNI 调用转换器，该转换器对每个所生成的 JIT 编译器中间表示指令执行分析，以寻找 JNI 函数调用。然后 JNI 调用转换器将同样多的 JNI 函数调用的中间表示转换为内部 JIT 编译器操作或常数，以便生成等价于 JNI 调用但不需要调用 JNI 函数的更简单的操作。以这种方式，提供了对 JVM 服务和数据的更快的访问。

为了发现程序打算调用的实际 JNI 方法，JNI 调用转换器首先必须理解每个 JNI 函数调用的 JIT 编译器中间表示的形状。为了理解每个形状，

JNI 调用转换器调用 JIT 编译器以依次编译调用每个 JNI 函数的不可执行的 C 或 C++程序。JNI 调用转换器获得关于每个 JNI 函数如何使用 JNIEnv 变元和每个调用的每个用户提供变元的使用的知识，并维护已知形状的列表。作为另一种选择，JIT 调用转换器可在 Java 程序执行的开始动态地确定形状或作为 JII 编译器构建过程的一部分确定形状。作为另一种选择，可将关于形状的知识硬编码到 JIT 编译器的源代码中。

JNI 调用的形状包括（1）访问实际目标 JNI 函数的手段（以中间表示表示的），（2）每个实际变元接收到的处理（同样以中间表示表示的）。典型的 JNI 调用的形状(当构建形状列表时被创建的)唯一地确定实际 JNI 例程。在该优选实施例中，构建形状列表（通过处理特殊的 C 和 C++程序）的过程发生在当该 Java 过程(并因此 JIT)开始执行时。也可能在构造 JIT 本身时创建形状列表。通过执行作为构建 JIT 编译器的一部分的形状确定，当编译器被重启时确定某一组形状。这样，用于每个 JNI 函数的形状可被静态地硬编码到 JIT 编译器中。为了执行作为构建 JIT 编译器的一部分的形状确定的唯一要求是所使用的中间表示对于 JIT 编译器的当前版本和虚拟机两者都是正确的。作为该过程的结果，生成 JNI 函数调用中间表示形状的列表。

一旦形状被确定，它们将由内嵌器使用，该内嵌器执行这样的内嵌工作，包括将参数映射为变元，合并本机编译器中间表示和 JIT 编译器中间表示，控制流图，和具体化 JNIEnv 指针从而内嵌的语句可使用它来进行对 JNI API 的调用。

在本机代码的内嵌期间，内嵌器分析内嵌的代码以发现所生成的代表 JNI 调用的中间表示语句。它通过将内嵌的中间表示匹配于从 C 或 C++程序生成的或在 JIT 构建过程中生成的形状的列表来进行上述分析。内嵌器记录在内嵌的本机函数中的、可被传递给递归内嵌的函数的 JNIEnv 变量的所有使用。当内嵌器遇到 JNIEnv 变量的一使用时，如果该 JNIEnv 变量未被用于与它在所述形状列表中的任何形状中所出现的相同的位置，则内嵌过程继续。然而，如果该 JNIEnv 变量被用于与它在所述形状列表中

的一个形状中所显示的相同的位置，则将该整个形状匹配于该 JNI 函数调用点（callsite）的中间表示。如果（一 JNI 方法调用的）形状与形状列表的一成员相比具有相同数量的变元，并且实际调用中的每个变元与形状列表中的该成员中的变元的类型一致，则该形状匹配形状列表中的该成员。如果发现匹配，这意味着该调用点对应于一 JNI 函数调用，则该调用点不适合于内嵌，但可适合于转换。

在这种情况下，JIT 编译器内嵌器记录它确定对应于 JNI 函数调用的调用点。生成这种 JNI 函数调用的一列表，该列表是按照在用于该内嵌方法的中间表示中出现的顺序排序的。如果一个方法调用使用另一个方法调用的结果，则最内的方法在该列表中将出现在该结果的使用之前。一旦完成了调用点的分析，则内嵌器识别不可内嵌的但可能可转换的 JNI 函数调用的列表。然后 JNI 调用转换器使用 JNI def/use 分析的结果对该 JNI 函数调用列表执行转换。

若干 JNI 函数调用被用于识别通常通过类的常数池来识别的元素。这些 JNI 函数的例子包括 getObjectClass、getMethodID、和 getFieldID。其他 JNI 函数调用可执行更具体的动作，例如获得或设置对象字段值。然而，在可执行优化之前必须知道被传递给 JNI 函数的对象的特定类型。因此，执行 JNI def/use 分析。

JNI def/use 分析类似于在优化编译器中经常使用的得到深入理解的定义/使用分析，它是通过跟踪通过内嵌的代码被传递给内嵌的本机方法的对象到它们作为变元被传递给 JNI 函数的位置来执行的。每个变元由一组可能的对象代表，该组对象取决于从内嵌的本机方法的开始到 JNI 函数的调用的控制流。该组对象可包括多于一个元素，并且这些元素可都来自相同的类。

def/use 分析计算到达 JNI 函数调用中的使用的所有可能的类、字段和方法。就是说，当它不能在一 JNI 调用点处决定性地确定例如一对象一定是其实例的类，则它产生足够的信息以允许该转换阶段考虑该对象可能是其实例的所有可能的类。

有可能 def/use 分析不能计算甚至有条件的结果。这将在例如一 JNI 函数调用的变元是从存储器中取出（已由先前的过程写在那里）以用于随后的 JNI 函数调用的情况下发生。

在 JNI def/use 分析中，getObjectClass、getSuperClass、findClass、getMethodID、和 getFieldID 被当作定义，并且它们的使用被跟踪。结果通常可被转换为将被使用以代替通常的常数池索引的常数值。

此外，JNI def/use 分析也跟踪 findClass、getMethodID、和 getFieldID 的串变元。以这种方式，JIT 编译器可确实地解析（resolve）这些调用中的某一些，而更天真的实现将不能这样做。然而，JNI def/use 分析的一个复杂之处是 NewObjectType JNI 调用的结果可到达在 JNI 函数调用的列表中的对象的多种使用。因此，这些结果具有一抽象类型，该抽象类型可在 JNI def/use 阶段期间用作 NewObjectType 的变元的类和 MethodID 变得固定的情况下变得具体。

如上所述，JNI 调用转换器使用 JNI def/use 分析的结果对 JNI 函数调用列表执行转换。在转换期间，JNI 调用转换器对 JNI 函数调用列表进行迭代，并将 JNI 函数调用替换为常数值，或生成新的更简单的 JIT 编译器中间表示并将 JNI 函数调用替换为该新的 JIT 编译器中间表示。

取决于 JNI 函数调用的类型，该转换过程的可能结果可以是不同的。例如，如果到达 getObjectClass 的所有可能的定义是同一类的，则该调用被替换为适当的常数。如下所述，在 def/use 分析产生包括关于到达 getObjectClass 的多个不兼容的（对象的）类的信息的已知结果的情况下，使用条件逻辑和代码复制（code replication）来使正确的实际类可用于随后的使用。

如果到达 getFieldID 或 getMethodID 的所有可能的类是兼容的（在 Java 的意义上），并且串变元可被唯一地确定，则该调用被替换为适当的常数。（如果不兼容的类的对象到达 getFieldID 或 getMethodID，则可如下所述插入条件逻辑。）

如果到达 get<type>Field 方法或 put<type>Field 方法（它们检索或存

储一 Java 类的字段）的所有可能的 fieldID 是相同的，并且到达该调用的所有可能的对象是兼容的类类型，则该调用被替换为相应的 JIT 编译器的中间表示的序列。应注意这种形式的内部表示根据用于执行本机方法的 Java 规则推迟了抛出任何异常。（表示法 get<type>Field 指意在从一对象检索字段的任何 JNI API 函数，并且<type>表示法的其他使用也类似。熟悉 JNI API 的技术人员将容易理解这种表示法。）

对于 JNI 函数列表中的各种 call<type>Method JNI 函数调用，通过移除被调用的方法的中间表示，并生成直接调用该函数的 JIT 中间表示，来执行类似的转换。对于 JNI 函数列表中不是使用以上步骤处理的任何条目，该调用被当作对适当的 VM 服务例程的普通调用。

重要的是注意到，对于上述任何函数，如果 JNI def/use 阶段产生了已知的但不是决定性的信息，则将基于传递到 JNI 函数调用中的实际接收器对象的类型或实际值（即，jfieldID 或 jmethodID）的条件逻辑与适当的中间表示一起插入以代表被转换的特定 JNI 函数的语义。

现转向图 5，该图示出了根据本发明的一优选实施例由 JNI 调用转换器执行的 JNI 函数调用转换。如图 5 所示，本发明提供了 JNI 调用转换器 500。

JNI 调用转换器 500 在由内嵌器内嵌的每个 JIT 编译器中间表示指令 502 中进行迭代。某些 JIT 编译器中间表示指令 502 可能是利用了 JNI API 函数的 JNI 函数调用 504。

对于那些是 JNI 函数调用的指令，JNI 调用转换器 500 将这些指令转换为转换的调用 506 和内部 JIT 编译器操作 508。转换的调用 506 可以是提供对 JVM 服务和数据的更直接和更快的访问的更简单的 JIT 编译器中间表示。内部 JIT 编译器操作 508 可以是常数值或不使用 JNI 的简化的中间表示。

以这种方式，JNI 调用转换器 500 修改某些或很多 JNI 函数调用以消除实际调用 JNI 函数的需要。该转换是通过使用内嵌器进行分析而将每个 JNI 函数的 JIT 编译器中间表示的形状匹配于已知形状的列表来完成的。

为了执行这种匹配，首先以如下在图 6 中描述的方式生成一形状列表。

现转到图 6，该图示出了根据本发明的一优选实施例为 JNI 函数调用生成中间表示形状。如图 6 中所示，JIT 编译器 600 编译包含对所有 JNI 函数的典型调用的不可执行的 C 和 C++程序 602 并生成代表用于 JNI 函数调用的中间表示的形状的列表 604。

现转到图 7，该图示出了根据本发明的一优选实施例用于 JNI 转换的本机内嵌准备。

如图 7 所示，JIT 编译器内嵌器 700 通过将用于本机调用的中间表示替换为 JIT 编译器中间表示格式的函数的实际实现来内嵌本机函数。

在内嵌期间，JIT 编译器内嵌器 700 可分析包括若干 JNI 函数调用的本机函数中间表示 702。在该示例中，本机函数中间表示 702 包括 JNI 调用 1、JNI 调用 2、JNI 调用 3、和 JNI 调用 4。对于由 JIT 编译器内嵌器匹配的每个 JNI 函数调用，JIT 编译器内嵌器 700 将该 JNI 函数调用以其在内嵌的本机函数的中间表示 702 中出现的顺序添加到 JNI 函数调用列表 704。

因此，JNI 调用 1 被添加到 JNI 函数调用列表 704，然后是 JNI 调用 2。如果一个 JNI 函数调用使用另一个 JNI 函数调用的结果，则最内的调用将在该表示中首先出现。

现转向图 8，该图示出了根据本发明的一优选实施例由编译器执行的示例性 def/use 分析。如图 8 所示，在 JNI def/use 分析中，编译器分析程序并确定是否可执行进一步的优化。

在该示例中，def/use 分析确定定义了什么变量。例如，程序 800 包括定义对象“obj” 802、整数“a” 804、类“cls” 806、和字段“fid” 808。此外，def/use 分析确定何处使用所定义的变量。例如，“obj”被用在 getObjectClass 方法 810 和 SetIntField 方法 814 中，“cls”被用在 getFieldID 方法 812 中，且“fid”被用在 setIntField 方法 814 中。因为“cls” 806 是 getObjectClass 方法 810 的结果，所以它被当作定义，且其使用被跟踪。在该图所示出的代码中，有可能将对 GetFieldID 的 JNI 函数调用替换为代

表该字段的 ID 的常数值。此外，因为“fid” 808 也是 setIntField 方法 814 的输入变元，将有可能将用于对 setIntField 方法 814 的调用的中间表示转换为直接修改对象中的相应字段的中间表示(即，无需进行 JNI 函数调用来这样做)。

现转向图 9，其示出了根据本发明的一优选实施例用于将 JNI 函数调用转换为常数、内部编译器操作或更简单的中间表示的流程图。该过程假设 JNI 调用转换器已如图 6 所示运行 JIT 编译器来生成 JIT 编译器中间表示和 JNI 函数调用形状列表。

如图 9 中所示，内嵌器被运行到其标准的内嵌阶段（步骤 900）。它查看它是否遇到任何可内嵌的本机调用点（步骤 902）。如果它没遇到，则它继续进行到步骤 926，该步骤重新开始 JIT 编译的其余部分。然而，如果遇到本机调用点，则它继续到步骤 904，在此它检查是否可内嵌的本机函数包含任何函数调用。如果不包含，则内嵌器继续进行到步骤 924，该步骤重新开始 JIT 编译的其余部分。如果可内嵌的本机调用确实包含函数调用，则该过程继续到步骤 906，在此将本机代码匹配于由图 6 中所示的过程生成的 JNI 函数调用形状。在步骤 908，检查调用点以发现匹配，且如果发现匹配，则步骤 910 将该调用点添加到 JNI 调用列表，并继续进行到步骤 912，该步骤为任何其他的调用点重复该最后过程。如果调用点不匹配，则该过程重复自身直到所有调用点已被检查。

一旦所有调用点已被分析，则步骤 914 执行 JNI def/use 分析，且通过在步骤 916 处理第一个被识别的 JNI 调用来继续。如果 JNI def/use 分析为该调用点产生了决定性的结果（步骤 918），则该调用的中间表示被替换为常数、更简单的编译器操作或更简单的中间表示，并可能带有用于已知结果的条件逻辑（步骤 922）。如果 JNI def/use 为该特定调用点生成了未知的结果，则从所述 JNI 函数调用列表中丢弃该调用点和任何其他使用其结果的调用点（步骤 920）。

一旦已使用 JNI def/use 的结果来转换该调用点，则该过程检查另外的 JNI 调用点（步骤 924）。如果有更多的 JNI 调用点，则该过程继续进行

到步骤 916。如果没有更多的调用点要处理，则步骤 926 继续进行 JIT 编译的其他部分，并且该过程随后结束。

重要的是注意到，尽管已在全功能的数据处理系统的情境中描述了本发明，本领域的普通技术人员将认识到本发明的过程也能够以指令的计算机可读介质的形式和多种形式被分发，并且不管实际用于执行这种分发的信号承载介质的特定类型是什么，本发明都同等适用。计算机可读介质的例子包括可记录类型的介质，例如软盘、硬盘驱动器、RAM、CD-ROM、DVD-ROM，以及传输类型的介质，例如使用诸如射频和光波传输等传输形式的数字和模拟通信链路、有线或无线的通信链路。该计算机可读介质可采取编码格式的形式，这种编码格式被解码以实际用于特定的数据处理系统。

本发明的描述是为了说明和描述的目的给出的，而非旨在是穷尽性的或限于本发明的所公开的形式。对本领域的普通技术人员来说很多修改和变形都将是显然的。例如，尽管所示的实施例是针对处理 Java 中的字节码，本发明的过程也可应用于处理这样的指令的其他编程语言和环境，所述指令不是特定于这些指令在其上执行的计算机。在这种情况下，该计算机上的虚拟机可解释这些指令或将这些指令发送给编译器以生成适于由该虚拟机位于其上的计算机执行的代码。

所选择和描述的实施例是为了最好地解释本发明的原理和实际应用，并使本领域的其他普通技术人员能理解本发明的带有适合于所考虑的特定应用的各种修改的各实施例。

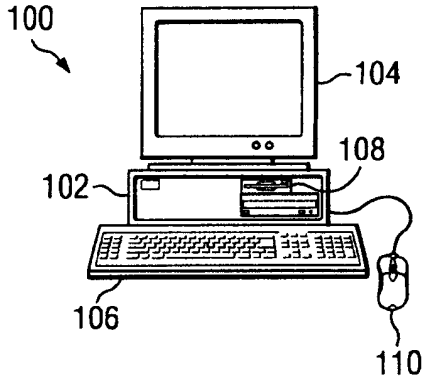

100
104
108
102
106
110

图 1

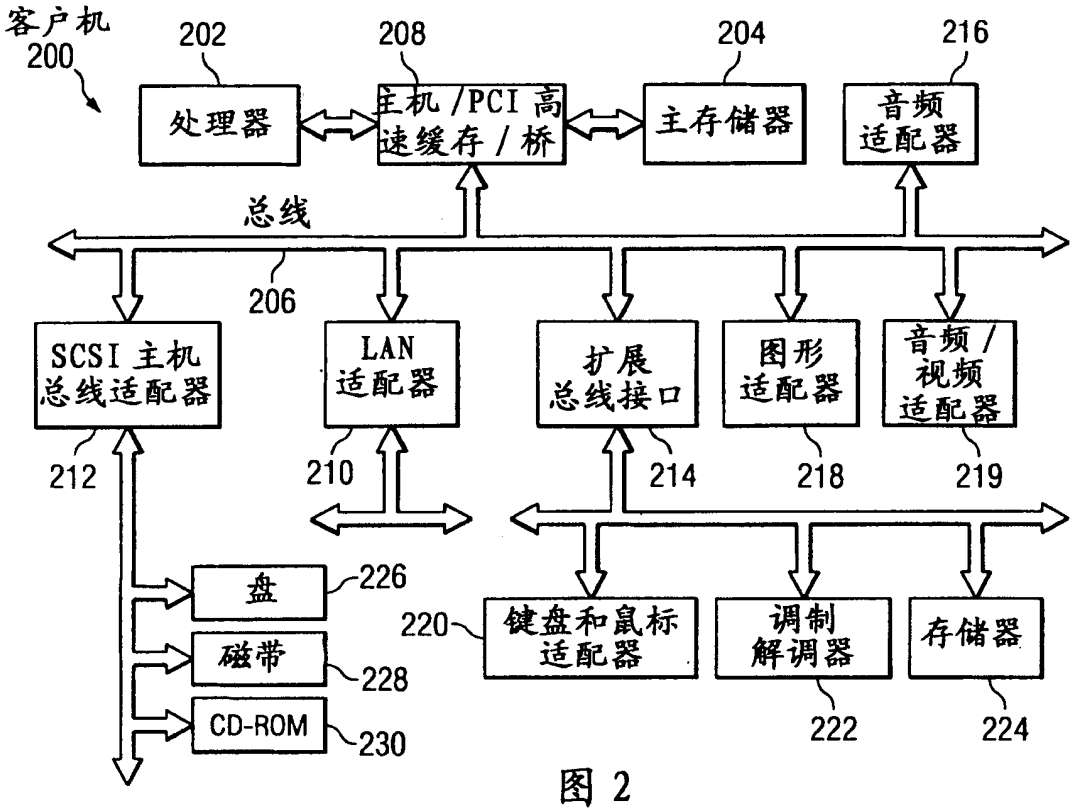

客户机
200
202
208
204
216
处理器
主机 /PCI 高速缓存 / 桥
主存储器
音频适配器
总线
206
SCSI 主机总线适配器
LAN 适配器
扩展总线接口
图形适配器
音频 / 视频适配器
212
210
214
218
219
盘
226
磁带
228
CD-ROM
230
220
键盘和鼠标适配器
调制解调器
存储器
222
224

图 2

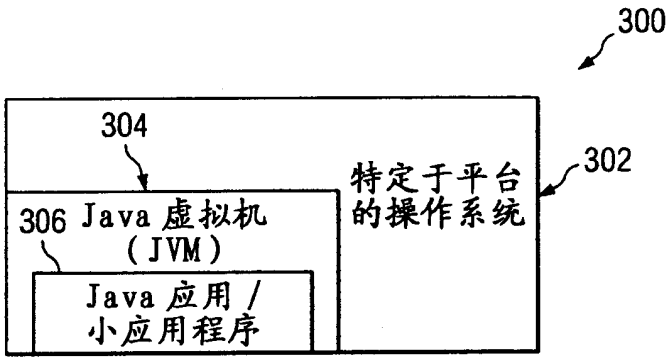

300
304
302
特定于平台
的操作系统
306 Java 虚拟机
(JVM)
Java 应用 /
小应用程序

图 3

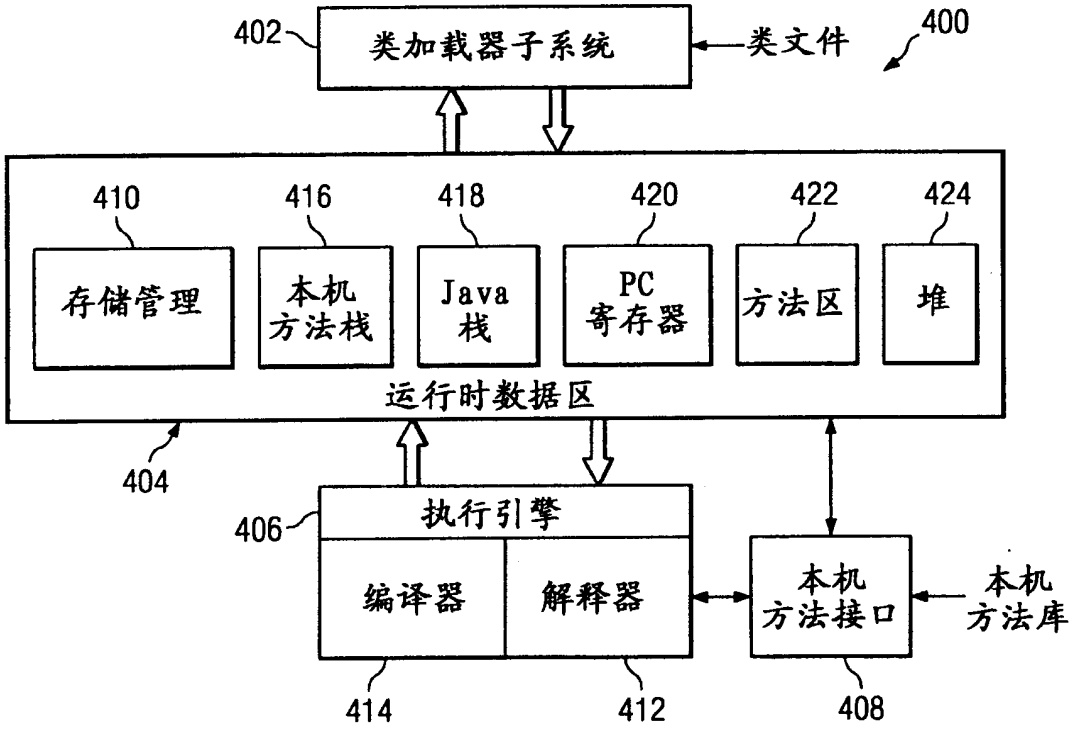

402
类加载器子系统
类文件
400
410
416
418
420
422
424
存储管理
本机
方法栈
Java
栈
PC
寄存器
方法区
堆
运行时数据区
404
406
执行引擎
编译器
解释器
本机
方法接口
本机
方法库
414
412
408

图 4

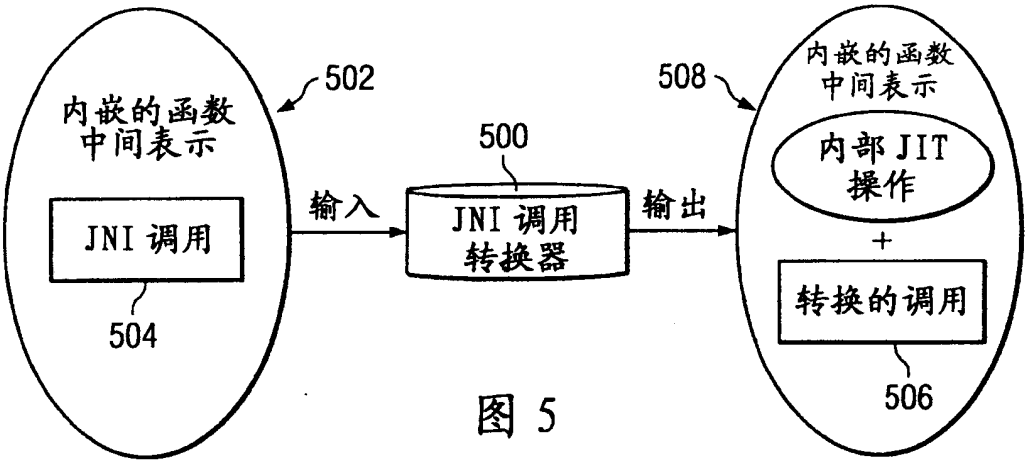

内嵌的函数
中间表示
502
JNI 调用
504
输入
500
JNI 调用
转换器
输出
508
内嵌的函数
中间表示
内部 JIT
操作
+
转换的调用
506

图 5

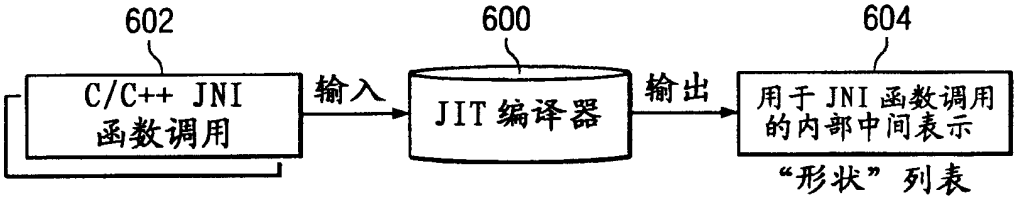

602
C/C++ JNI
函数调用
输入
600
JIT 编译器
输出
604
用于 JNI 函数调用
的内部中间表示
“形状”列表

图 6

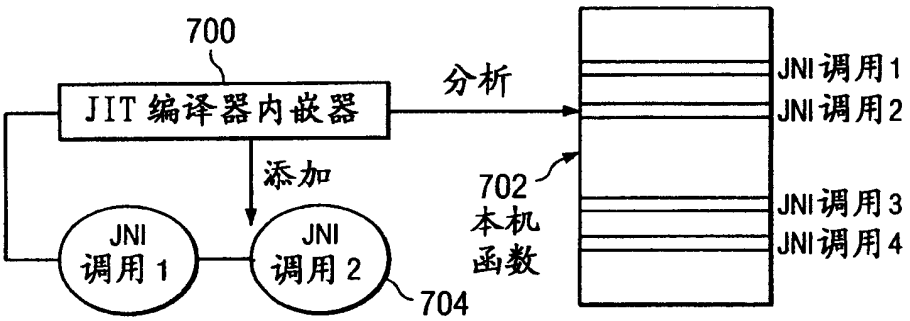

700
JIT 编译器内嵌器
分析
添加
JNI
调用 1
JNI
调用 2
704
702
本机
函数
JNI调用1
JNI调用2
JNI调用3
JNI调用4

图 7

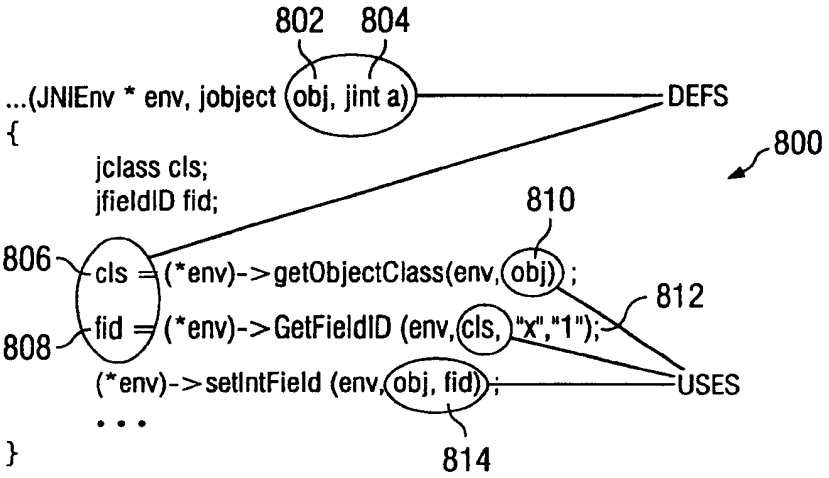

802
804
...(JNIEnv * env, jobject obj, jint a)
DEFS
{
jclass cls;
jfieldID fid;
800
810
806
cls = (*env)->getObjectClass(env, obj);
812
808
fid = (*env)->GetFieldID (env, cls, "x", "1");
(*env)->setIntField (env, obj, fid);
USES
...
}
814

图 8

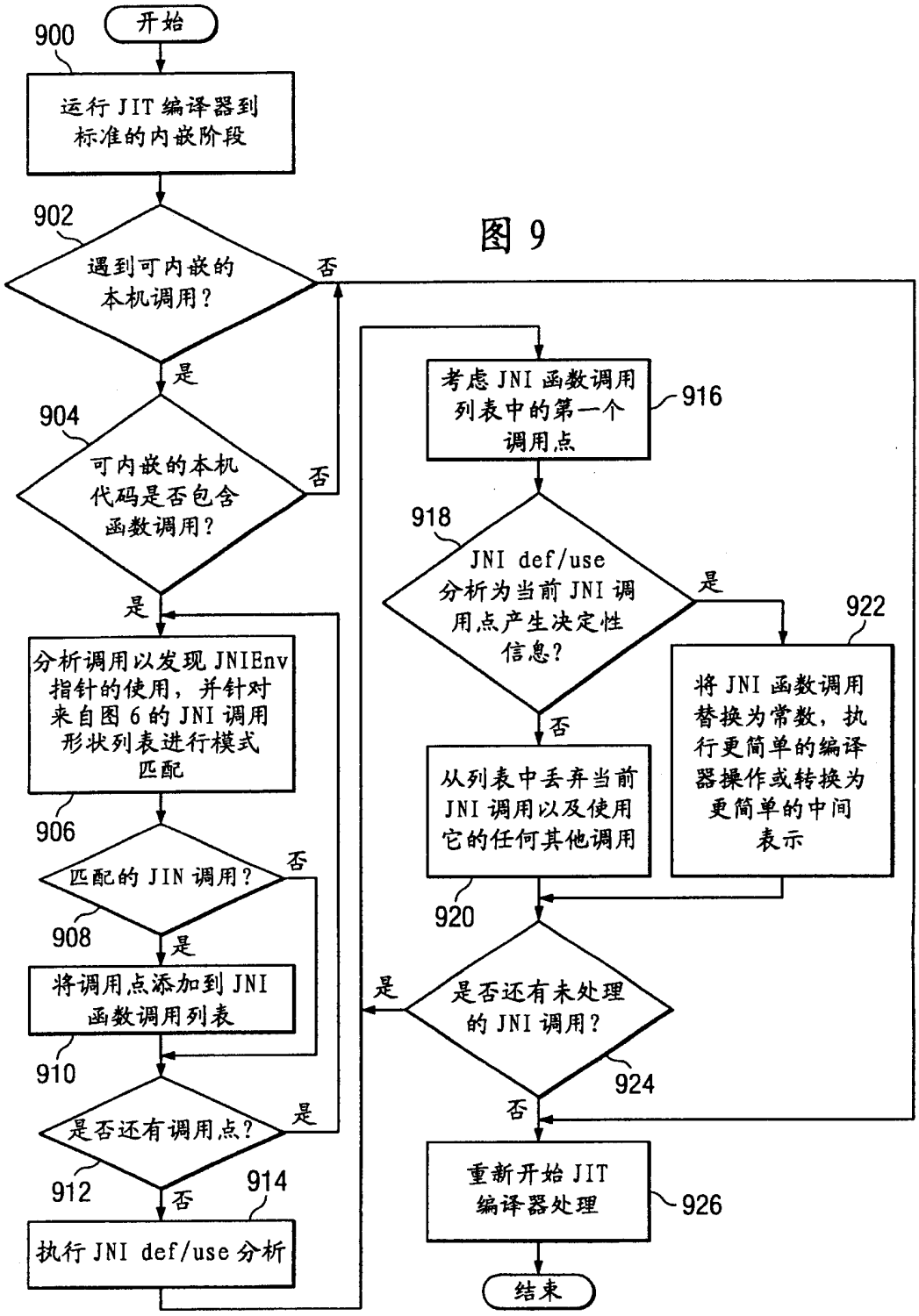

开始
900
运行 JIT 编译器到标准的内嵌阶段
902
遇到可内嵌的本机调用?
否
是
904
可内嵌的本机代码是否包含函数调用?
否
是
分析调用以发现 JNIEnv 指针的使用，并针对来自图 6 的 JNI 调用形状列表进行模式匹配
906
匹配的 JIN 调用?
否
908
是
将调用点添加到 JNI 函数调用列表
910
是否还有调用点?
是
912
否
914
执行 JNI def/use 分析
考虑 JNI 函数调用列表中的第一个调用点
916
918
JNI def/use 分析为当前 JNI 调用点产生决定性信息?
是
922
将 JNI 函数调用替换为常数，执行更简单的编译器操作或转换为更简单的中间表示
否
从列表中丢弃当前 JNI 调用以及使用它的任何其他调用
920
是否还有未处理的 JNI 调用?
是
924
否
重新开始 JIT 编译器处理
926
结束

图 9