



(12) 发明专利申请

(10) 申请公布号 CN 102200911 A

(43) 申请公布日 2011. 09. 28

(21) 申请号 201110080598. 6

(22) 申请日 2011. 03. 23

(30) 优先权数据

12/730, 263 2010. 03. 24 US

(71) 申请人 微软公司

地址 美国华盛顿州

(72) 发明人 I·任科夫斯基 H·坎恩塔姆那尼

(74) 专利代理机构 上海专利商标事务所有限公司 31100

代理人 顾嘉运

(51) Int. Cl.

G06F 9/44 (2006. 01)

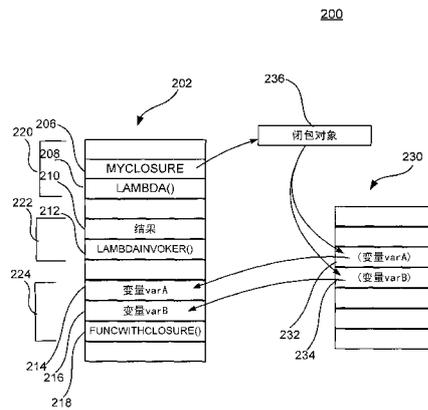
权利要求书 2 页 说明书 11 页 附图 6 页

(54) 发明名称

变量闭包

(57) 摘要

描述了一种用于从特定程序配置中的拉姆达表达式来访问闭包变量的系统和方法,其中当闭包函数是活动时访问闭包变量,且闭包变量位于与闭包函数的实例相对应的活动运行时框架栈之中。响应于进入闭包函数,将指向每个闭包变量的指针存储在栈指针表中。通过使用闭包变量指针来定位运行时栈上的闭包变量,来处理从拉姆达表达式对每个闭包变量的每个引用。程序代码可被插装以便在进入闭包函数的位置处以及在从拉姆达表达式对每个闭包变量的每个引用处插入对助手函数的调用。



1. 一种基于计算机的方法,所述方法用于当计算机程序的函数的实例(218)活动时,从调用自所述函数之外的计算机程序位置(206)的拉姆达表达式访问在所述函数中定义的闭包变量,所述变量被存储在运行时栈(202)上的框架(214)中,所述框架对应于所述函数实例,所述方法包括:

a) 响应于在程序执行期间进入所述函数(304),将指向所述变量的指针存储(306)在数据结构中;

b) 响应于提取所述变量的值的指令,所述拉姆达表达式中的指令使用所述变量指针来定位(404)运行时栈(202)上的框架(214)中的变量并提取(406)变量值。

2. 如权利要求1所述的基于计算机的方法,其特征在于,还包括生成对应于所述函数实例的闭包对象;以及使用所述闭包对象来定位所述变量指针。

3. 如权利要求1所述的基于计算机的方法,其特征在于,还包括响应于对所述变量进行赋值的指令,所述拉姆达表达式中的指令使用所述变量指针来定位所述运行时栈上的框架中的变量并将所述值存储在所述变量中。

4. 如权利要求1所述的基于计算机的方法,其特征在于,还包括自动将一个或多个指令插入函数以便于响应于在程序执行期间进入所述函数而存储指向所述变量的指针。

5. 如权利要求1所述的基于计算机的方法,其特征在于,还包括自动将一个或多个指令插入所述拉姆达表达式以便于使用所述变量指针来定位所述运行时栈上的框架中的变量。

6. 如权利要求1所述的基于计算机的方法,其特征在于,还包括插装所述函数以便于响应于进入所述函数而存储指向所述变量的指针;以及插装所述拉姆达表达式以便于定位所述运行时栈上的框架中的变量。

7. 如权利要求1所述的基于计算机的方法,其特征在于,还包括:

a) 自动插入闭包创建指令以响应于对所述函数的调用而创建闭包对象;

b) 自动插入变量引用指令以从所述拉姆达表达式中提取闭包变量值;

c) 提供程序代码以创建所述闭包对象;以及

d) 提供程序代码以提取所述闭包变量值。

8. 如权利要求1所述的基于计算机的方法,其特征在于,所述函数包括引用所述变量的指令,所述方法还包括不插装所述指令,以允许所述指令访问所述运行时栈上的框架中的变量。

9. 一种基于计算机的系统,所述系统用于访问在计算机程序的闭包函数中定义的闭包变量,所述系统包括:

a) 代码插装组件(108),其被配置为执行以下动作:

i. 解析(502)所述函数并插入(506)程序代码以将指向所述闭包变量的指针存储在数据结构中;

ii. 解析(508)拉姆达函数以确定对所述闭包变量的一个或多个引用,并插入(510)指令以访问运行时栈框架中的与所述函数的实例相对应的闭包变量;

b) 执行以下动作的第一助手函数(112):提取指向所述闭包变量的指针,使用所述指针来定位(404)所述运行时栈框架中的所述闭包变量,并从所述闭包变量的位置中提取(406)值;以及

c) 执行以下动作的第二助手函数 (112) : 存储 (306) 指向所述程序变量的指针。

10. 如权利要求 9 所述的基于计算机的系统, 其特征在于, 存储指向所述闭包变量的指针的程序代码包括调用所述第二助手函数的指令, 所述指令被插入在所述函数中。

11. 如权利要求 9 所述的基于计算机的系统, 其特征在于, 插入访问所述闭包变量的指令包括将对所述闭包变量的一个或多个引用中的每一个均替换为调用所述第一助手函数的指令。

12. 如权利要求 9 所述的基于计算机的系统, 其特征在于, 包括用于以下功能的装置: 使所述闭包函数中的拉姆达表达式能够访问所述闭包变量而不在堆存储器中创建所述闭包变量的副本。

13. 如权利要求 9 所述的基于计算机的系统, 其特征在于, 还包括响应于对所述函数实例的调用而创建与每一个函数实例相对应的闭包对象的程序代码, 所述闭包对象包括用于提取指向所述闭包变量的指针的引用。

14. 如权利要求 9 所述的基于计算机的系统, 其特征在于, 还包括用于使用所述指针来将另一个值存储在所述运行时栈框架中的所述闭包变量中的装置。

15. 如权利要求 9 所述的基于计算机的系统, 其特征在于, 还包括用于使多个拉姆达函数实例能够访问所述运行时栈框架中的所述闭包变量的装置。

变量闭包

技术领域

[0001] 本发明涉及基于计算机的方法和系统,尤其涉及访问计算机程序函数中定义的闭包变量。

背景技术

[0002] 一些计算机编程语言包括被称为闭包的概念。闭包是捕捉诸如程序变量之类的环境元素以供后续使用的的一种方式,即便原始元素已经改变或不再存在了。例如,内函数可以引用在外函数内定义的局部变量。内函数可分配到变量并被调用。概念上,内函数所引用的变量可以在赋值的时候被保存,并且这些已保存的变量可以在内函数的后续调用期间使用。一种实施方式可以将变量保存在所分配的堆存储器中,并在不再需要这些变量时解除这些堆存储器的分配。对内函数的多次调用引用所保存的变量的相同副本。引用外函数的局部变量的内函数被称为拉姆达 (lambda) 函数,或简称为拉姆达。

发明内容

[0003] 提供本发明内容是为了以简化的形式介绍将在以下具体实施方式中进一步描述的一些概念。本发明内容并不旨在标识出所要求保护的的主题的关键特征或必要特征,也不旨在用于限定所要求保护的的主题的范围。

[0004] 简要地,一种系统、方法和组件用于当对多个闭包变量中的一个的引用是在计算机程序的闭包函数的拉姆达表达式中时,访问在该闭包函数中定义的闭包变量。闭包变量可以是对闭包函数而言局部的并被存储在运行时栈上的栈框架中,该栈框架对应于闭包函数的一个实例。响应于进入闭包函数,可以将指向每一个闭包变量的指针存储在诸如堆存储器或其他类型的存储器之类的数据结构中。响应于提取变量值的指令(该指令在拉姆达表达式中),可以使用变量指针来定位运行时栈上的框架中的变量,并提取该变量值。

[0005] 在一个实施方式中,响应于对闭包函数的实例的调用,可以生成对应于该实例的闭包对象,该闭包对象包括指向栈指针的指针。

[0006] 在一个实施方式中,可以通过解析计算机程序并插入指令以便于提取闭包变量而插装 (instrument) 该计算机程序。这可包括:插入指令以创建闭包对象,并在调用该闭包函数时存储变量指针。这可包括插入指令以响应于拉姆达表达式中的引用而提取闭包变量。这可进一步包括插入指令以便在退出闭包函数时删除对象。

[0007] 为了实现前述及相关目的,在这里结合以下描述及附图来描述该系统的某些说明性方面。然而,这些方面仅指示了可采用本发明的原理的各种方法中的少数几种,且本发明旨在包括所有这样的方面及其等效方面。通过结合附图考虑本发明的以下具体实施方式,本发明的其它优点以及新颖的特征将变得显而易见。

附图说明

[0008] 参考下述附图描述了本发明的非限制性且非穷尽性实施方式。在这些附图中,相

同的附图标记指代各附图中的相同部分,除非另外指明。

[0009] 为了帮助理解本发明,将参考与附图相关联地阅读的具体实施方式,在附图中:

[0010] 图 1 是采用在此描述的机制的计算机系统的框图。图 2 是可被用于实现在此描述的机制中的至少某一些的一组数据结构。

[0011] 图 3 是示出实现变量闭包的过程的示例实施方式的流程图。图 4 是示出对拉姆达函数中的闭包变量进行处理的过程的示例实施方式的流程图。

[0012] 图 5 是用于插装程序代码以便于图 3 和 4 的过程的流程图;

[0013] 图 6 示出了计算机设备的一个实施方式,示出了可被用于执行在此描述的功能的计算机设备的所选组件。

具体实施方式

[0014] 下文中将参考附图来更全面地描述本发明的各示例实施方式,附图构成实施方式的一部分且在其中作为示例示出了可在其中实践本发明的各特定示例实施方式。然而,本发明可被实现为许多不同的形式并且不应被解释为被限于此处描述的各实施方式;相反,提供这些实施方式以使得本公开变得透彻和完整,并且将本发明的范围完全传达给本领域技术人员。特别地,本发明可被实现为方法或设备。因此,本发明可采用完全硬件实施方式、完全软件实施方式或者结合软件和硬件方面实施方式的形式。因此,以下详细描述并非是局限性的。

[0015] 贯穿说明书和权利要求书,下列术语采用此处显式相关联的含义,除非该上下文在其他地方另有清楚指示。如此处所使用的,短语“在一个实施方式中”尽管它可以但不一定指前一实施方式。此外,如此处所使用的,短语“在另一个实施方式中”尽管它可以但不一定指前不同的实施方式。因此,可以容易地组合本发明的各实施方式而不背离本发明的范围或精神。类似地,如此处所使用的,短语“在一个实现中”尽管它可以但不一定指相同的实现,并且可以组合各种实现的技术。

[0016] 另外,如此处所使用的,术语“或”是包括性“或”运算符,并且等价于术语“和/或”,除非上下文清楚地另外指明。术语“基于”并非穷尽性的并且允许基于未描述的其他因素,除非上下文清楚地另外指明。另外,在本说明书全文中,“a”、“an”和“the”的含义包括复数引用。“在……中”的含义包括“在……中”和“在……上”。

[0017] 此处所描述的组件可以从其上具有数据结构的各种计算机可读介质来执行。组件可通过本地或远程过程诸如按照具有一或多个数据分组(例如,来自一个通过信号与本地系统、分布式系统中的另一组件交互或跨网络诸如因特网的其它系统交互的组件的数据)的信号来通信。例如,软件组件可被存储在非瞬态计算机可读存储介质上,包括但不限于根据本发明的各实施方式的以下计算机可读存储介质:专用集成电路(ASIC)、紧致盘(CD)、数字多功能盘(DVD)、随机存取存储器(RAM)、只读存储器(ROM)、软盘、硬盘、电可擦除可编程只读存储器(EEPROM)、闪存或记忆棒。

[0018] 如此处所用的术语“计算机可读介质”既包括非瞬态存储介质又包括通信介质。通信介质一般用诸如载波或其他传输机制等已调制数据信号来体现计算机可读指令、数据结构、程序模块或其他数据,并且包括任何信息传递介质。作为示例而非限制,通信介质包括诸如有线网络和直接线连接等有线介质,以及诸如声学、无线电、红外线和和其他无线介质等

无线介质。

[0019] 如此处所使用的,术语“应用程序”指计算机程序或其一部分并且可包括相关联的数据。应用程序可以是独立程序或者应用程序可被设计成向另一应用程序提供一个或多个特征。“附加件”和“插件”是与“主机”应用程序交互并向其提供特征的应用程序的示例。

[0020] 应用程序由应用程序组件的任何组合构成,应用程序组件可包括程序指令、数据、文本、对象代码、图像或其他媒体、安全证书、脚本、或者可被安装在计算设备上以使该设备能够执行所需功能的其他软件组件。应用程序组件能够以文件、库、页面、二进制块或数据流的形式存在。

[0021] 如此处所使用的,术语“指针”指对目标物理或逻辑存储器位置、数据结构、程序指令或程序分段的引用。指针“指向”目标并且可用于定位或获取目标。指针能够以各种方式实现,包括地址、偏移量、索引或标识符。

[0022] 如此处所使用的,除非上下文另外指明,否则术语“功能”指执行特定任务的较大程序中的一部分代码,并且能够相对独立于该程序的其他部分执行。功能可以但不一定返回值。在各种计算机语言中,可使用不同的术语,诸如子例程、方法、过程或子程序。如此处所使用的,术语“功能”可以包括所有这些。

[0023] 图 1 是其中可实现在此所描述的机制的计算机系统 100 的框图。图 1 只是合适的系统配置的一个示例,并且不旨在对本发明的使用范围或功能提出任何限制。因此,可采用各种系统配置而不背离本发明的范围或精神。

[0024] 如图所示,系统 100 包括程序源代码 102,该程序源代码可以是计算机程序的高级语言表示。高级语言的示例包括 F-Sharp (F#)、Visual Basic 或各种其他高级语言。作为语言和库扩展的组合物 LINQ 是程序源代码 102 的另一示例。在被编译成本机代码之前被编译成 IL(中间语言)的语言有时被称为“托管语言”。程序可包括一个或多个函数。程序可以驻留在一个或多个文件或其他存储表示中。程序可包括一个或多个库,该一个或多个库能够以各种方式集成或分布。因此,程序源代码 102 可表示程序库或其一部分。

[0025] 如图所示,系统 100 包括编译器前端 104。在一个实施方式中,编译器前端包括词法分析器、句法分析器、以及语义分析器,但可以使用各种其他组件或配置。在一个实施方式中,编译器前端 104 处理程序源代码 102,将其转换成中间语言模块 106。在一个实现中,中间语言模块 106 可表示整个程序源代码 102,并包括多个函数,但也可包括程序源代码 102 的仅仅一部分或函数的一部分。在一个实现中,中间语言模块 106 被存储为一个或多个文件。在一个实现中,中间语言模块 106 包括对应于程序源代码 102 的二进制指令序列或二进制流。

[0026] 虽然没有示出,但一个实施方式中该系统可以包括运行时管理器,运行时管理器是管理计算机程序的执行的系统组件。在各种配置中,运行时管理器可执行多个动作中的一个或多个,包括加载通过执行计算机程序来调用的程序函数、翻译程序函数、定位并加载程序所采用用的库或其他资源、或调用或管理各种程序资源。运行时管理器可被成为实现向正在执行的计算机程序提供各种资源和服务的系统框架。

[0027] 在一个配置中,运行时管理器包括即时 (JIT) 编译器或其一部分。通常,JIT 编译器采用一机制,其中响应于第一次调用,程序函数的中间语言表示被加载并转换成本机语言表示。例如,当正在运行的程序第一次调用函数时,响应于检测到该调用,函数的中间语

言表示可被快速编译为本机代码并随后运行。本机语言表示可被存储在存储器中,以便对于后续调用不需要进行翻译。运行时管理器的一个示例是华盛顿州雷德蒙市的微软公司制作的公共语言运行时 (CLR) 组件。CLR 组件使用被称为公共中间语言 (CIL) 的中间语言表示。在一个配置中,运行时管理器的 JIT 编译器可响应于检测到对程序或函数的调用而紧靠在执行之前将 IL 转换为本机代码。在一个实施方式中,系统可使用多个进程,以使得 JIT 编译器可包括在执行另一个函数的同时加载或转换函数的进程。系统可在执行调用前检测到对函数的调用,以使得在执行调用前执行加载或转换的至少一部分。术语“检测”包括在执行调用前在运行时期间检测调用。在一个配置中,运行时管理器可在运行时之前将 IL 转换为本机代码。

[0028] 如图 1 所示,代码插装组件 (CIC) 108 可从 IL 模块 106 接收函数,并执行多种变换,诸如将指令插入特定位置。修改可包括:添加、删除、移动、或修改程序指令。插入或修改程序指令的过程被称为“插装”。该进程,以及这些修改的示例,将在下文中进行详细描述。

[0029] 系统 100 可包括链接器 110,该链接器执行组合并链接程序函数、修改或插入变量或函数引用等的各种操作。在一个实施方式中,链接器 110 可提取一个或多个助手函数 112,并将这些函数与中间语言程序进行组合以产生链接程序。

[0030] 系统 100 可包括代码生成器 114,该代码生成器将中间代码表示转换为本机代码 116。本机代码 116 可以是机器语言、虚拟机语言、或可以由物理或虚拟处理器执行的另一表示。处理器 120 可以提取本机代码 116 并执行程序指令,以产生执行结果 122。在一个配置中,处理器 120 可包括一个或多个中央处理单元、一个或多个处理器核、ASIC、或其他硬件处理组件及相关程序逻辑。在一个配置中,处理器 120 可包括仿真硬件处理单元的软件组件。处理器 120 执行本机代码 116 形式的指令。

[0031] 执行结果 122 是执行本机代码 116 的结果的逻辑表示。结果可包括对计算机存储或计算机存储器的修改、与其他进程或计算设备的通信、音频或视频输出、或对各种系统或外部组件的控制中的一个或多个。

[0032] 系统 100 可以是开发系统的子系统。开发系统可包括由作为程序开发、测试、或文档编制的一部分的程序开发者或用户采用的一个或多个计算设备。系统 100 的组件可分布在一个或多个计算设备中,这些计算设备中的每一个可通过使用各种有线或无线通信协议(诸如 IP、TCP/IP、UDP、HTTP、SSL、TLS、FTP、SMTP、WAP、蓝牙、WLAN 等等)中的一种或多种与其他计算设备进行通信。

[0033] 计算设备可以是专用或通用计算设备。简要地,可以被使用的计算设备的一个实施方式包括一个或多个处理单元、存储器、显示器、键盘和定点设备、以及通信接口。示例计算设备包括大型机、服务器、刀片服务器、个人计算机、便携式计算机、通信设备、消费电子产品等等。计算设备可包括通用或专用操作系统。华盛顿州雷德蒙市的微软公司制造的 Windows® 系列操作系统是可以在开发系统的计算设备上执行的操作系统的示例。

[0034] 图 1 仅仅是适合的系统的一个示例,并且不旨在提出对本发明的使用范围或功能的任何限制。因此,可使用各种系统配置而不背离本发明的范围或精神。例如,CIC 108 或链接器 110 可以与编译器前端 104 组合。一些系统可直接转换为本机代码而不需要中间语言。可以使用多种其他配置。

[0035] 表 1 包括示出变量闭包的示例的代码片断。

[0036] 表 1

示例闭包	
L100	<code>static void Main(string[] args) {</code>
L101	<code> Func<int> lambda = FuncWithClosure();</code>
L102	<code> int result = lambda();</code>
L103	<code>}</code>
L104	<code>static Func<int> FuncWithClosure() {</code>
L105	<code> int varA = 100;</code>
[0037] L106	<code> int varB = 200;</code>
L107	<code></code>
L108	<code> Func<int> lambda = () => {</code>
L109	<code> return varA + varB;</code>
L110	<code>};</code>
L111	<code> varA = 150;</code>
L112	<code> return lambda;</code>
L113	<code>}</code>

[0038] 在该代码片断中，每一行都开头标有带有前缀“L”的行号，以避免与在此所使用的附图标号相混淆。在该示例性代码片断中，将创建一闭包，该闭包包括外函数 `FuncWithClosure()` 的变量 `varA` 和 `varB`。外函数可以被称为是“闭包函数”。行 L108 到行 L110 创建一将被绑定到闭包的拉姆达表达式。更具体而言，行 L108 到行 L110 定义了拉姆达函数，其是拉姆达表达式的一种类型。术语“拉姆达”在此指拉姆达表达式。

[0039] 行 L108 到 L110 的拉姆达使用闭包变量 `varA` 和 `varB`。语句 L112 返回拉姆达至 `FuncWithClosure()` 的调用者。语句 L101 调用外函数，接收由调用创建的对拉姆达的引用。在语句 L102，调用拉姆达，返回值 350。具体而言，可以注意到拉姆达使用已保存的变量环境，并在行 L111 反映对局部变量 `varA` 的修改。即便外函数 `FuncWithClosure()` 在行 L102 处调用拉姆达时不再活动，已保存的环境变量仍然继续存在。

[0040] 在闭包的一个实现中，由拉姆达函数访问的外函数的局部变量被复制到堆存储器中的一位置内。从拉姆达函数或外函数对这些变量的引用将被定向至该堆存储器位置。由拉姆达函数访问的外函数的局部变量被称为“封闭 (closedover)”变量。术语“封闭变量”和“闭包变量”是同义词并且在此可以互换使用。

[0041] 表 2 包括示出闭包的另一个示例的示例源代码片断。表 2 的示例与表 1 的不同之处在于拉姆达是在外函数仍然活动时调用的，并由此包括封闭变量的外函数的局部变量仍然处于活动栈框架中。

[0042] 表 2

示例闭包

[0043]

L200	static void Main(string[] args) {
L201	FuncWithClosure();
L202	}
L203	static void FuncWithClosure() {
L204	int varA = 100;
L205	int varB = 200;
L206	
L207	Func<int> lambda = () => {
L208	return varA + varB;
L209	};
L210	varA = 150;
L211	lambdaInvoker(lambda);
L212	}
L213	static void lambdaInvoker(Func<int> lambda) {
L214	int result = lambda();
L215	}

[0044] 在该示例代码片断中，将创建一闭包，该闭包包括外函数 FuncWithClosure() 的变量 varA 和 varB。行 L207 到行 L209 创建一将被绑定到该闭包的拉姆达表达式。拉姆达使用变量 varA 和 varB。语句 L210 在创建拉姆达之后修改 varA。语句 L201 调用外函数。语句 L211 调用函数 LambdaInvoker()，将拉姆达传递给该函数。LambdaInvoker() 中的语句 L214 调用拉姆达，其引用 varA 的修改值并返回值 350。在该代码片断中，当拉姆达在语句 L214 被调用时，外函数 FuncWithClosure 对于运行时栈上的活动框架是活动的。

[0045] 可以注意到在该代码片断中，拉姆达函数和外函数两者均引用相同的 varA。因此，外函数对 varA 的改变导致拉姆达函数使用改变的值。

[0046] 表 3 包括示出用于实现表 2 中的代码片断的闭包的机制的示例源代码片断。在表 3 中，插入或修改指令以实现其中封闭元素仍旧位于运行时栈上的变量闭包。注意，表 3 的示例代码片断概念性地示出了插装过程的一个实施方式，但实际插装结果可以是有所不同的。

[0047] 表 3

具有插装的示例闭包

```

L300 static void Main(string[] args) {
L301     FuncWithClosure();
L302 }
L303 static void FuncWithClosure() {
L304     int varA, varB;
L305     Closure closure = RuntimeHelpers.CreateClosure(ref varA, ref
L306     varB);
L307     try {
L308         varA = 100;
L309         varB = 200;
L310         Func<int> lambda =
[0048] RunTimeHelpers.CreatelambdaWithClosure(closure, myClosure => {
L311             return (int)myClosure.GetVariable(0) +
L312             (int)myClosure.GetVariable(1);
L313         });
L314         varA = 150;
L315         lambdaInvoker(lambda);
L316     } finally {
L317         RuntimeHelpers.DeleteClosure(closure);
L318     }
L319 static void lambdaInvoker(Func<int> lambda) {
L320     int result = lambda();
L321 }

```

[0049] 如表 3 所示的, 在一个实施方式中, 运行时助手对象被用于执行实现闭包的操作。例如, 行 L305 示出了创建闭包对象的对运行时助手函数的插装调用, 向其传递将要被封闭的每个局部变量的地址。在行 L316, 在退出外函数之前, 插入删除闭包对象的对助手函数的匹配的插装调用。外函数, 即 FuncWithClosure() 的主体, 被放置于“try”子句中 (行 L306), 并且用于删除闭包的插装调用被放置于“finally”子句中 (行 L315), 由此, 即便当函数由于异常退出时闭包对象也被删除。

[0050] 在一个实施方式中, 助手函数被用于创建拉姆达对象。在表 3 中, 行 L309 包括对该助手函数的插装调用。助手函数 GetVariable() 被用于提取每个封闭变量。在该示例中, 每个变量均有一索引值, 并且引用变量是通过传入对应的索引值来执行的。在该示例中, 闭包变量 varA 和 varB 分别具有索引值 0 和 1。

[0051] 在表 3 所示的实施方式中, 行 312 处的外函数对封闭变量 varA 的引用不被插装。这允许该语句以与非封闭的局部变量相类似的方式引用外函数栈框架中的封闭变量。

[0052] 图 2 是可以被用于实现处理表 2 的示例代码片断的机制的数据结构 200 的框图。如上所述, 在一个实施方式中, 表 2 的代码片断可以如表 3 的代码片断所概念性地表示的那样被插装。图 2 对应于表 3 的插装代码片断。

[0053] 在图 2 中,数据结构包括运行时栈 202、栈指针表 230、以及闭包对象 236。运行时栈 202 包含栈条目 206-218。每个栈条目属于与活动函数的实例相对应的栈框架。在图 2 的示例中,栈条目 214、216 和 218 属于与 FuncWithClosure() 的实例相对应的栈框架 224;栈条目 210 和 212 属于与 lambdaInvoker() 的实例相对应的栈框架 222;栈条目 206 和 208 属于与 lambda() 的实例相对应的栈框架 220。虽然所示出的条目用于示出在此描述的至少一些机制的各方面,但运行时栈可包括更多或更少的栈框架,以及更多或更少的栈条目。

[0054] 如图所示,栈框架 224 包括包含变量 varA 的数据的栈条目 214、包含变量 varB 的数据的栈条目 216、以及包含 FuncWithClosure() 的其他数据的栈条目 218。栈框架 222 包括包含变量结果的数据的栈条目 212 以及包含函数 lambdaInvoker() 的其他数据的栈条目 212。栈框架 220 包括包含拉姆达对象“myClosure”的栈条目 206,拉姆达对象“myClosure”指向闭包对象 236 并被用于引用封闭变量 varA 和 varB。

[0055] 栈指针表 230 包括对应于每个闭包之内的元素的数据。具体而言,栈指针条目 232 包括指向栈条目 214(varA) 的指针而栈指针条目 234 包括指向栈条目 216(varB) 的指针。

[0056] 在所示出的实施方式中,闭包对象 236 与 FuncWithClosure() 的闭包以及与拉姆达函数相关联。其包括指向栈指针条目 232 和 234(分别对应于闭包变量 varA 和 varB) 的指针。

[0057] 在下面的讨论中,对表 3 的各行的引用将被用于指定各操作与表 3 的插装代码之间的对应关系。一个实施方式以如下方式运作。在运行时期间,当进入实例 FuncWithClosure() 并将对应的框架 224 推送到运行时栈 202 上时,调用创建闭包的运行时助手函数(行 L305)。栈指针条目 232 和 234 被加入栈指针表 230,每个条目包含指向栈条目 214 或 216 的指针,栈条目 214 或 216 分别对应于闭包的局部变量。同样,创建带有指向栈指针条目 232 和 234 的指针的闭包对象 236。在一个实施方式中,可以通过调用助手函数来创建拉姆达(行 L309)。所得拉姆达被绑定到闭包对象 236,这使得当调用拉姆达时,闭包对象 236 被传递至拉姆达(在该示例中作为“myClosure”自变量)。

[0058] 当随后调用拉姆达函数时,诸如在行 L320,使用闭包对象 236 来分别从栈条目 214 和 216 中提取变量 varA 和 varB 的值。这可通过如下步骤来执行:使用闭包对象中的变量索引,从栈指针条目 232 和 234 中提取相应的指针,并跟着栈指针到所述栈条目。由此,虽然遵从了闭包的语义,但在运行时栈 202 上而不是在堆存储器块中维护局部变量 varA 和 varB 的数据。在一些配置中,这种机制可避免对于闭包变量废弃(garbage)堆存储器的集合,并减少各种资源的执行时间或其他使用。

[0059] 虽然图 2 示出了单个拉姆达函数,但在具有多个引用封闭变量 varA 和 varB 的拉姆达函数实例的程序中,在一个实施方式中每个实例可具有指向闭包对象 236 的拉姆达对象,以使得每个引用访问运行时堆栈 202 上的同一个对应变量的值。例如,上述情况会在如下情况下发生:在表 2 的行 L202-L208 处的拉姆达函数调用第二拉姆达函数,并且两个拉姆达函数均是对于对应的堆栈框架而活动的。在另一个示例中,上述情况会在如下情况下发生:一个或多个拉姆达函数的多个实例被顺序地调用,以使得在第二实例被推送到栈上之前从运行时栈内移除一个实例。

[0060] 图 3 是示出实现变量闭包的过程 300 的示例实施方式的流程图。在一个实施方式中,过程 300 的一些动作由图 1 的计算机系统 100 的组件来执行。

[0061] 过程 300 所示出的部分可以在判定框 302 处启动,在那里可确定对闭包的变量的访问是否被限于变量范围的生存期。在一个实施方式中,判定框 302 的判定可例如通过开发程序代码的程序员手动进行。在一个实施方式中,该判定可诸如由计算机程序自动进行,该计算机程序分析程序代码以确定是否满足合适条件。

[0062] 在判定框 302,如果判定是否定的,则该过程进行到框 318,在那里使用不同于在此所描述的机制的机制来实现闭包。该过程可退出或返回到调用程序。

[0063] 在判定框 302,如果判定是肯定的,则该过程进行到框 304,在那里检测对具有变量闭包的函数的调用。该函数在此被称为“外”函数或闭包函数。如在此所讨论的,在一个实施方式中,框 304 的动作可包括对助手函数的使用,所插入的插装程序代码调用调用该助手函数来指示何时调用外函数。

[0064] 该过程可进行到框 306,在那里将指向闭包的变量的指针放置在指定位置。一个这样的位置可以是栈指针块,诸如表 2 的栈指针表 230。一个实现可包括对应于每个闭包的每个封闭变量的条目。

[0065] 该过程进行到框 308,在那里创建对应于外函数闭包的闭包对象,以使得闭包对象引用框 306 所描述的指针。各种实现可包括对闭包的封闭变量指针的集合的单次引用,以及启用每个封闭变量指针的配置的对应配置。一些实现可包括从闭包对象到栈指针表的多次引用。

[0066] 该过程进行到框 310,在那里处理拉姆达函数中的一个或多个闭包变量引用,以便将每次引用与运行时栈上的对应的封闭变量相关联。框 310 的动作在图 4 中示出,并且将在下文中更详细地描述。

[0067] 该过程进行到框 312,在那里检测外部闭包函数的退出。如在此所讨论的,这可通过在退出外部闭包函数之前插装程序代码以包括助手函数调用来促进。响应于此检测,该过程可进行到框 314,在那里可删除闭包对象,并且可从栈指针表中移除封闭变量指针。可以理解,在此所使用的删除可由多种方式来实现,包括标记将被后续进程删除的块。

[0068] 该过程可进行到完成框 316,并退出或返回到调用程序。图 3 的过程 300 可针对于多个闭包同时或顺序地执行一次或多次。

[0069] 图 4 是用于处理拉姆达函数中的闭包变量引用的过程 400 的流程图。在一个实施方式中,过程 400 或其变形可以执行图 3 的块 310 的至少一部分动作。

[0070] 过程 400 从循环 402 开始,该循环为每次对封闭变量的引用而进行迭代。例如,表 2 的行 L208 包括对封闭变量 varA 和 varB 的引用。在一个实施方式中,循环 402 可对变量 varA 迭代一次并且对变量 varB 迭代一次。在其中行 L207 的拉姆达被调用多次的实施方式中,循环 402 可对每次调用迭代两次。在所示实施方式中,循环 402 包括框 404 和 406,并在框 408 处终止。

[0071] 在框 404,基于闭包变量指针和闭包对象来确定封闭变量在运行时栈上的位置。参考图 2 的示例结构,在一个实现中,变量 varA 在运行时栈上的位置被指示为栈条目 214。闭包对象 236 可被用于定位栈指针条目 232,其是指向变量 varA 的闭包变量指针。

[0072] 该过程可进行到框 406,在那里访问运行时栈上的封闭变量。该访问可以是读访问或写访问。在表 2 的行 L208 的示例中,在循环 402 的一次迭代期间提取变量 varA 的值,并在循环 402 的另一次迭代期间提取变量 varB 的值。一些计算机程序可以包括将值存储在

封闭变量中的指令。在这样的情况下,框 406 的动作可包括将值存储在运行时栈上的封闭变量中。

[0073] 该过程可进行到框 408,在那里终止循环 402。在循环 402 的最后一次迭代之后,过程 400 可进行到完成框 410,并返回到调用程序,诸如图 3 的过程 300。

[0074] 图 5 是示出插装程序代码以便于过程 300 和 400 的过程 500 的流程图。在一个实现中,通过插入对方便在此描述的机制的助手函数的调用来插装该程序代码。

[0075] 过程 500 的所示出的部分可在框 502 通过解析程序代码开始。程序代码可具有源代码、中间语言、或本机代码的形式。这可包括确定闭包函数的进入或退出位置,拉姆达函数的位置、或对封闭变量的引用。虽然解析动作是在框 502 中示出的,但在各种实现中,解析程序代码可结合过程 504 的其他动作在各部分中执行。因此,图 5 所示出的每个框可包括一些程序代码解析,或可在每一个框之前执行一定量的解析。

[0076] 该过程进行到框 504,在那里在外闭包函数的开始和结尾中插入语句,以指示该函数的调用和退出。如过程 300 所描述的,该语句可包括调用助手函数以创建闭包对象或将指向闭包变量的指针放置在栈指针表中并且删除该数据。在表 3 中,行 L305 提供了用于创建对应于变量 varA 和 varB 的闭包的插装函数调用的示例。行 L316 提供了用于在退出外函数之前删除闭包的插装函数调用的示例。

[0077] 该过程进行到框 506,在那里可插入程序代码以创建拉姆达对象。行 L309 提供了创建拉姆达对象的对助手函数的插装调用的示例。

[0078] 该过程进行到循环 509,其开始为拉姆达对象中的对闭包变量的每一次引用进行迭代的循环。例如,表 2 的行 L208 包括两个这样的引用:对 varA 的引用以及对 varB 的引用。

[0079] 该过程进行到框 510,在那里可以用访问运行时栈上的闭包变量的程序代码来替换当前迭代中的变量引用。在一个实现中,这可包括在运行时期间执行该动作的对助手函数的函数调用。访问闭包变量可包括提取变量值或将值存储在变量中。

[0080] 该过程进行到框 512,在那里终止循环 508。当完成循环 508 的所有迭代时,该过程可进行到完成框 514,并且退出或返回到调用程序。

[0081] 图 6 示出计算设备 600 的一个实施方式,示出了可以被用于执行在此所描述的功能(包括过程 300、400 或 500)的计算设备的所选组件。计算设备 600 可包括比所示出的要多得多的组件,或可包括比所示出的全部组件要少的组件。计算设备 600 可以是独立的计算设备或是集成系统的一部分,诸如是带有一个或多个刀片的机箱中的一个刀片。

[0082] 如所示出的,计算设备 600 包括一个或多个处理器 602,其进行动作以执行各种计算机程序的指令。在一个配置中,每个处理器 602 可包括一个或多个中央处理器、一个或多个处理器核、一个或多个 ASIC、高速缓存存储器、或其他硬件处理组件以及相关程序逻辑。如所示出的,计算设备 600 包括操作系统 604。操作系统 604 可以是通用或专用操作系统。华盛顿州雷德蒙市的微软公司制造的 Windows® 系列操作系统是可在计算设备 600 上执行的操作系统的示例。

[0083] 存储器 606 可包括各种类型的非瞬态计算机存储介质中的一个或多个,包括易失性或非易失性存储器、RAM、ROM、固态存储器、盘驱动器、光存储、或可被用于存储数字信息的任何其他介质。

[0084] 存储器 606 可存储在此描述的一个或多个组件或其他组件。在一个实施方式中，存储器 606 存储 CIC 108、运行时栈 202、助手函数 112、以及栈指针表 230。这些组件中的一个或多个可以被操作系统 604 或其他组件移动到 RAM、非易失性存储器中的不同位置，或在 RAM 和非易失性存储器之间移动。

[0085] 计算设备 600 可包括方便向用户显示本地化文本串的视频显示适配器 612，以及将文本转换为音频语音并向用户呈现语音串的语音组件（没有示出）。虽然没有示出在图 6 中，但计算设备 600 可包括基本输入 / 输出系统 (BIOS) 以及相关联的组件。计算设备 600 也可包括网络接口单元 610 以用于与网络进行通信。计算设备 600 的实施方式可包括显示监视器 614、键盘、定点设备、音频组件、话筒、语音识别组件、或其他输入 / 输出机制中的一个或多个。

[0086] 可以理解，图 3-5 的流程图示中的每一个框，以及这些流程图示中的框的组合都可由软件指令来实现。这些程序指令可被提供给处理器以产生一机器，以使得在处理器上执行的指令创建用于实现一个或多个流程图框内指定的动作的装置。软件指令可以由处理器执行以提供用于实现在一个或多个流程图框内指定的动作的步骤。此外，流程图示中的一个或多个框或框的组合也可与其他框或框的组合一起同时被执行，或甚至以与所示的顺序不同的顺序被执行，而不背离本发明的范围或精神。

[0087] 以上说明、示例和数据提供了对本发明的组成部分的制造和使用的全面描述。因为可以在不背离本发明的精神和范围的情况下做出本发明的许多实施方式，所以本发明落在所附权利要求的范围内。

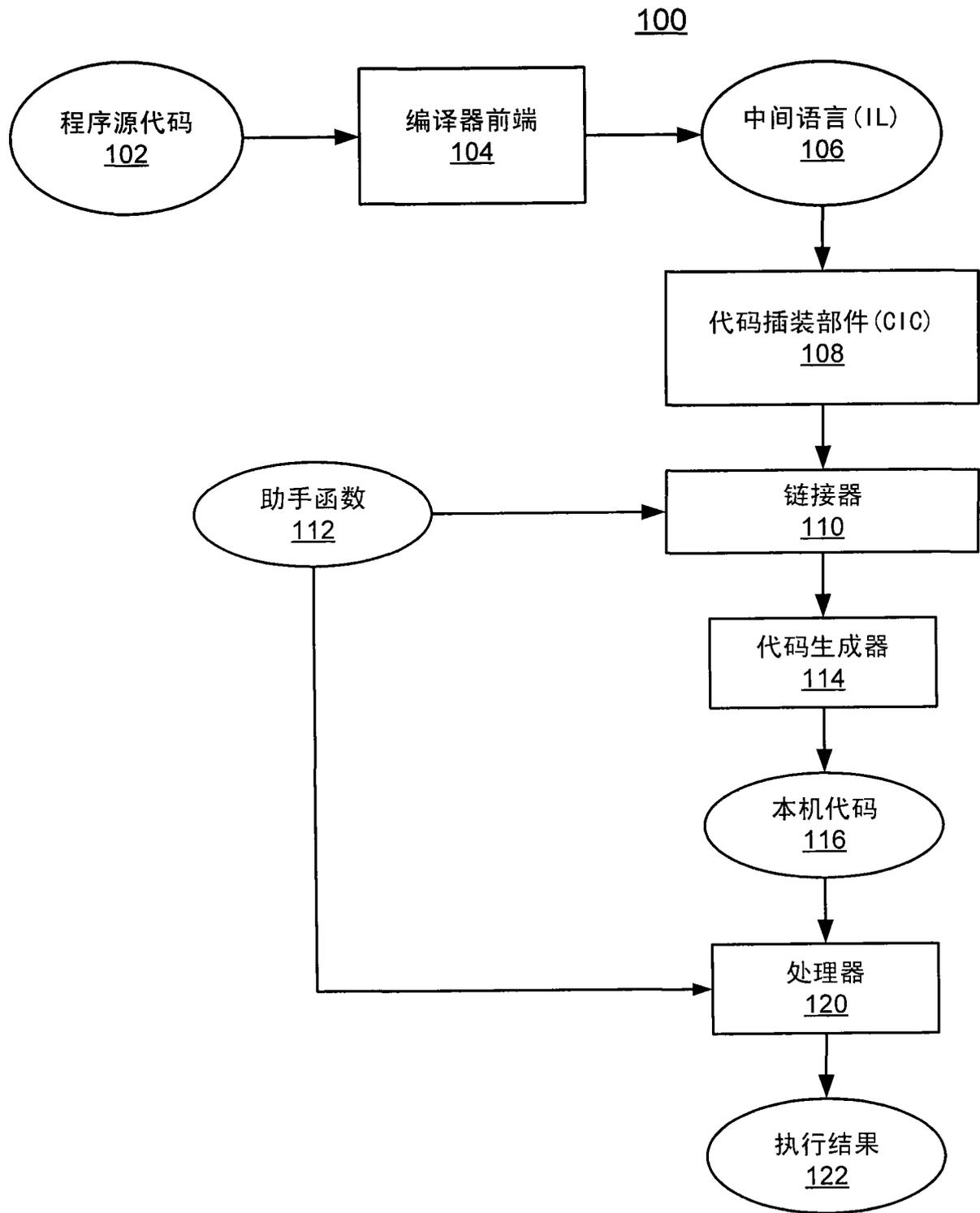


图 1

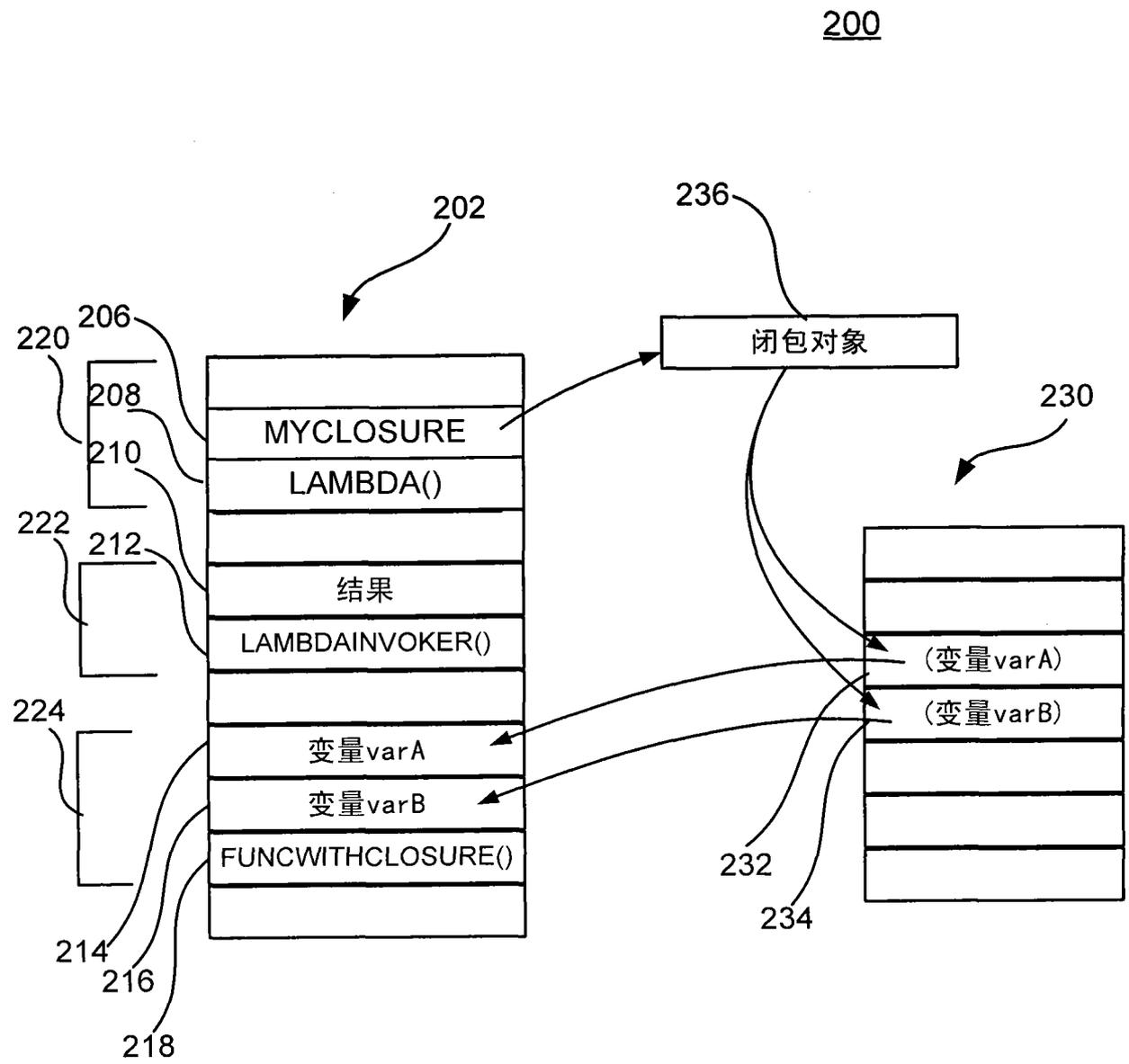


图 2

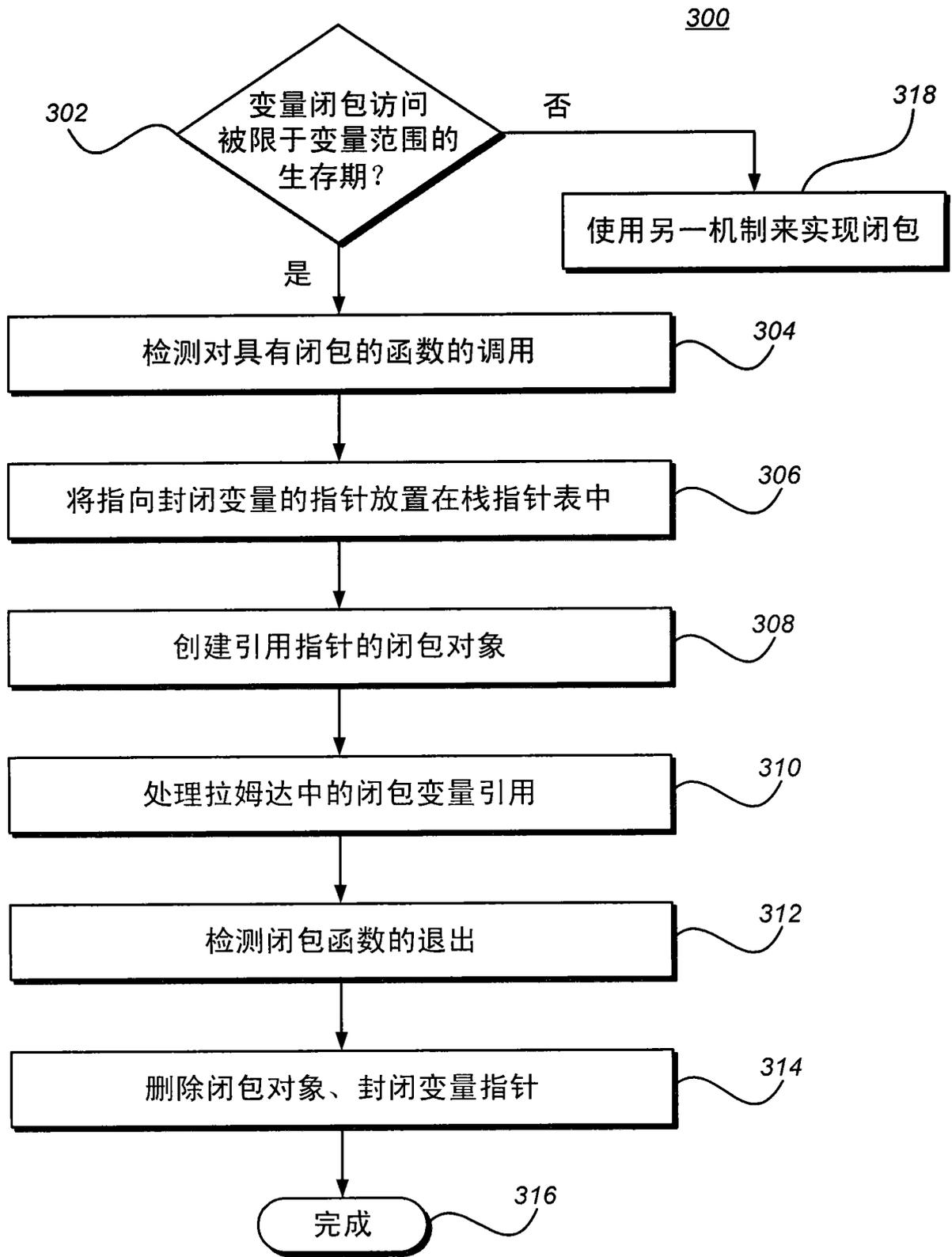


图 3

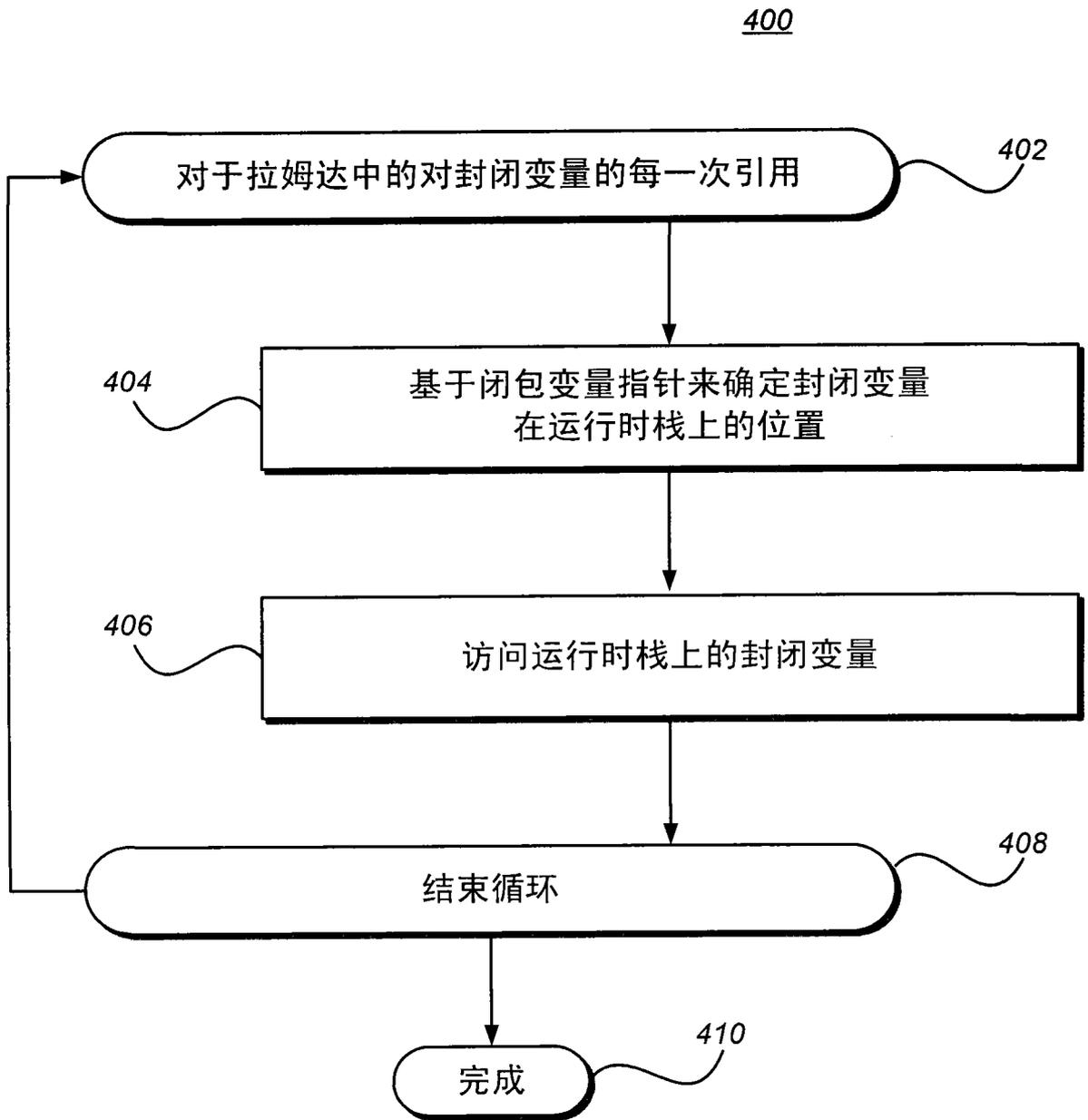


图 4

500

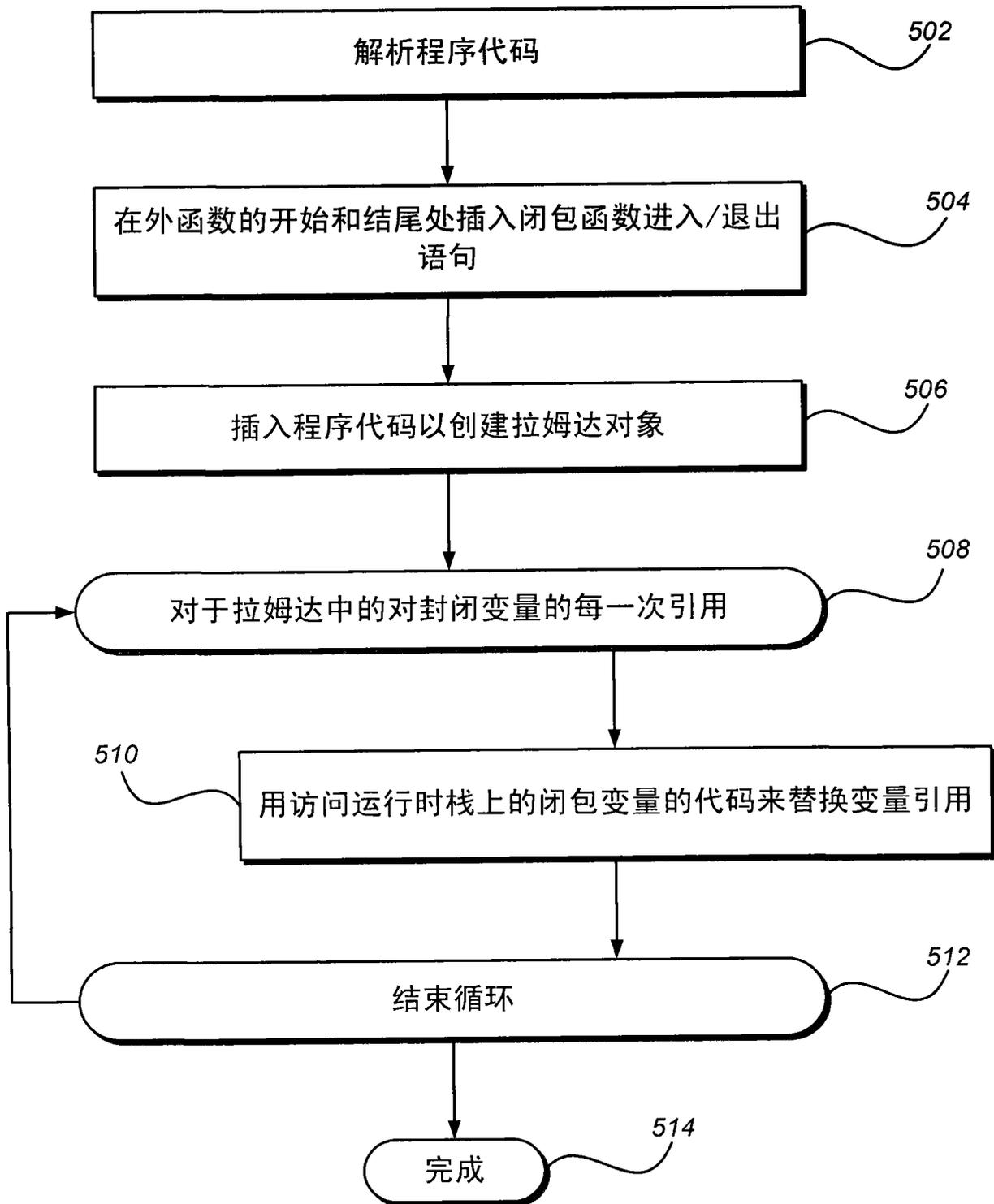


图 5

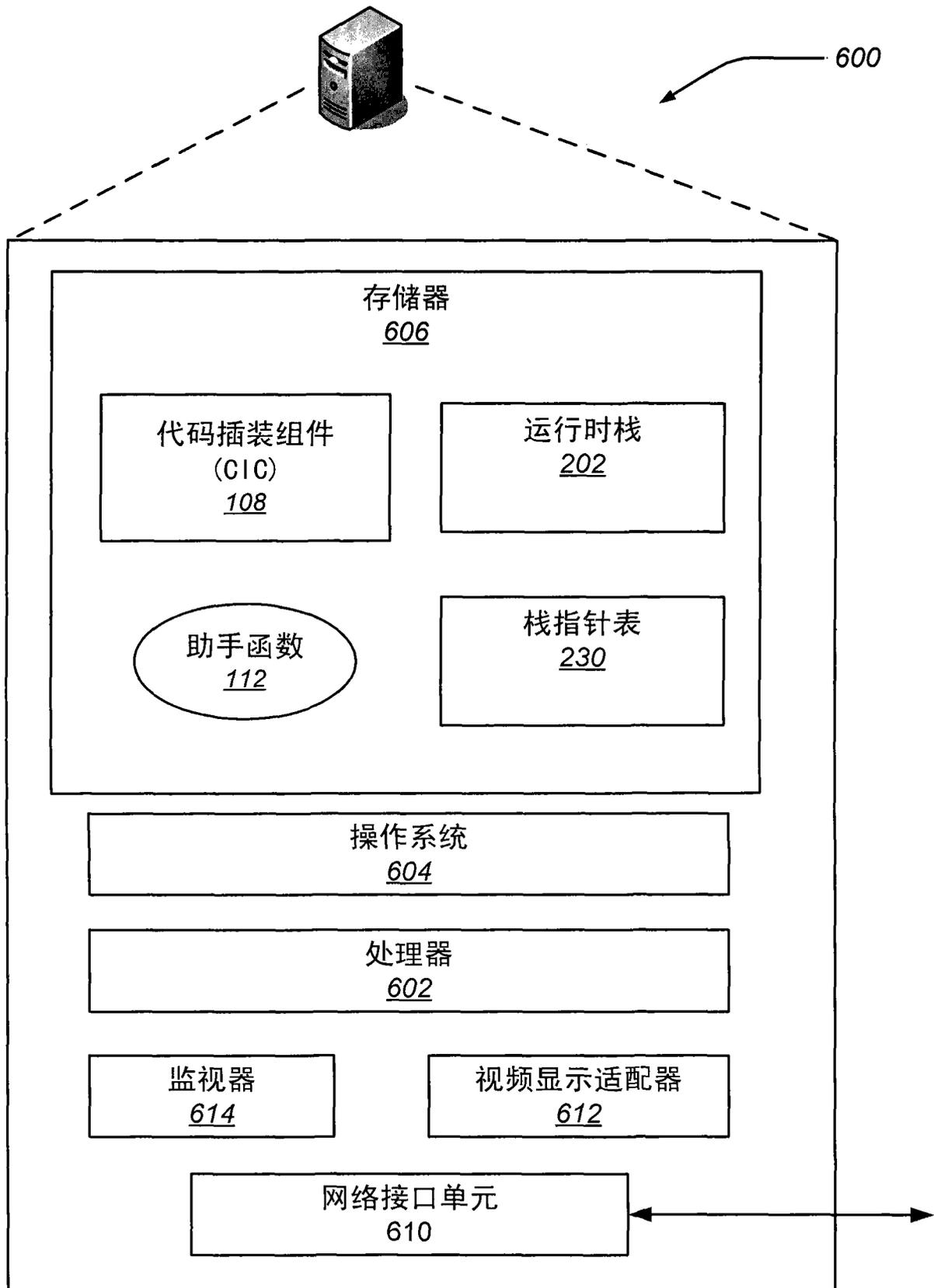


图 6