(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 104471557 B
(45)授权公告日 2016.11.02

(21)申请号 201380038085.6
(22)申请日 2013.06.17
(65)同一申请的已公布的文献号
申请公布号 CN 104471557 A
(43)申请公布日 2015.03.25
(30)优先权数据
13/526,328 2012.06.18 US
(85)PCT国际申请进入国家阶段日
2015.01.16
(86)PCT国际申请的申请数据
PCT/US2013/046164 2013.06.17
(87)PCT国际申请的公布数据
WO2013/192104 EN 2013.12.27
(73)专利权人 谷歌公司
地址 美国加利福尼亚州
(72)发明人 所罗门·布洛斯 杰里米·休格曼
(74)专利代理机构 中原信达知识产权代理有限责任公司 11219
代理人 周亚荣 安翔
(51)Int.Cl.
G06F 15/16(2006.01)
(56)对比文件
US 8095507 B2,2012.01.10,
US 2009031292 A1,2009.01.29,
CN 101923492 A,2010.12.22,
CN 101076793 A,2007.11.21,
US 5530964 A,1996.06.25,
审查员 俞立文

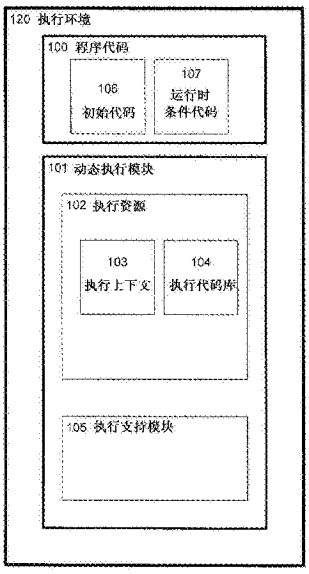
权利要求书2页 说明书14页 附图4页

(54)发明名称

动态语言的优化执行

(57)摘要

通过包括执行上下文和执行代码库的动态执行模块来执行动态语言的程序代码。程序代码的指令被转换为存储在执行代码库中的机器代码，执行上下文通过执行以及跨越执行跟踪程序的运行时状态和持久状态。使用执行代码库和执行上下文，动态执行模块使重复的机器代码生成最小化，同时保留了程序的动态方面。



CN 104471557 B

1.一种用于通过动态执行模块执行计算机程序的计算机实现的方法,所述方法包括:

接收执行以动态语言的程序代码的第一请求,所述程序代码包括初始代码和运行时条件代码;

响应于所述第一请求,从所述初始代码生成执行资源,所述执行资源包括执行代码库和执行上下文,所述执行代码库包括从所述初始代码生成的初始机器代码,以及所述执行上下文包括存储的对针对所述初始代码中的函数、变量和用户定义类型生成的元素的引用;

使用所述执行上下文来执行所述执行代码库中的所述初始机器代码;

响应于执行所述执行代码库中的所述初始机器代码,使用所述运行时条件代码来更新所述执行代码库和所述执行上下文以在所述执行代码库中存储从所述运行时条件代码生成的条件机器代码并且将所述运行时条件代码和代码上下文映射至存储的条件机器代码;

接收执行所述程序代码的第二请求;以及

当当前代码上下文对应于所述代码上下文时,使用更新的执行上下文来执行所述初始机器代码并且执行更新的执行代码库中的所述条件机器代码,否则至少基于所述运行时条件代码生成并且执行更新的条件机器代码,并且将所述运行时条件代码和所述当前代码上下文映射至所述执行代码库中的更新的条件机器代码。

2.根据权利要求1所述的计算机实现的方法,其中,使用所述执行上下文来执行所述执行代码库中的所述初始机器代码还包括使用存储的引用来访问针对函数、变量和用户定义类型生成的元素。

3.根据权利要求1所述的计算机实现的方法,其中,所述运行时条件代码包括函数,并且所述代码上下文包括所述函数利用其被调用的至少一个包含。

4.根据权利要求1所述的计算机实现的方法,其中,所述执行上下文还包括包含存储的对元素的引用的函数表、变量表和类表。

5.根据权利要求4所述的计算机实现的方法,其中,更新所述执行上下文还包括在所述函数表、变量表和类表中存储对针对所述运行时条件代码中的函数、变量和类生成的元素的引用。

6.根据权利要求5所述的计算机实现的方法,其中,所述执行上下文还包括源表示库,以及更新所述执行上下文还包括将所述运行时条件代码的一部分的表示存储在所述源表示库中。

7.根据权利要求6所述的计算机实现的方法,其中,所述执行上下文还包括包含表,以及更新所述执行上下文还包括在所述包含表中生成条目,所述条目包括到所述源表示库中的所述运行时条件代码的所述部分的第一引用和到所述执行代码库中的所述条件机器代码第二引用。

8.根据权利要求1所述的计算机实现的方法,其中,所述执行上下文还包括分配表,所述分配表包括针对所述程序代码中的所述变量和用户定义类型分配的元素。

9.根据权利要求1所述的计算机实现的方法,其中,所述程序代码、所述初始代码和所述运行时条件代码包括PHP计算机指令。

10.一种用于通过动态执行模块执行计算机程序的系统,所述系统包括:

用于接收执行以动态语言的程序代码的第一请求的装置,所述程序代码包括初始代码

和运行时条件代码；

用于响应于所述第一请求，从所述初始代码生成执行资源的装置，所述执行资源包括执行代码库和执行上下文，所述执行代码库包括从所述初始代码生成的初始机器代码，以及所述执行上下文包括存储的对针对所述初始代码中的函数、变量和用户定义类型生成的元素的引用；

用于使用所述执行上下文来执行所述执行代码库中的所述初始机器代码的装置；

用于响应于执行所述执行代码库中的所述初始机器代码，使用所述运行时条件代码来更新所述执行代码库和所述执行上下文以在所述执行代码库中存储从所述运行时条件代码生成的条件机器代码的装置；

用于将所述运行时条件代码和代码上下文映射至所述执行代码库中的存储的条件机器代码的装置；

用于接收执行所述程序代码的第二请求的装置；以及

用于当当前代码上下文等同于所述代码上下文时，使用更新的执行上下文来执行所述初始机器代码并且执行更新的执行代码库中的所述条件机器代码，否则至少基于所述运行时条件代码生成并且执行更新的条件机器代码，并且将所述运行时条件代码和所述当前代码上下文映射至所述执行代码库中的更新的条件机器代码的装置。

11.根据权利要求10所述的系统，其中，用于使用所述执行上下文来执行所述执行代码库中的所述初始机器代码的装置还包括用于使用存储的引用来访问针对函数、变量和用户定义类型生成的元素的装置。

12.根据权利要求10所述的系统，其中，用于更新所述执行代码库的装置还包括用于将从所述运行时条件代码生成的所述机器代码存储在所述执行代码库中的装置。

13.根据权利要求12所述的系统，其中，所述执行上下文还包括包含存储的对元素的引用的函数表、变量表和类表。

14.根据权利要求13所述的系统，其中，用于更新所述执行上下文的装置还包括用于在所述函数表、变量表和类表中存储对针对所述运行时条件代码中的函数、变量和类生成的元素的引用的装置。

15.根据权利要求14所述的系统，其中，所述执行上下文还包括源表示库，以及用于更新所述执行上下文的装置还包括用于将所述运行时条件代码的一部分的表示存储在所述源表示库中的装置。

16.根据权利要求15所述的系统，其中，所述执行上下文还包括包含表，以及用于更新所述执行上下文的装置还包括用于在所述包含表中生成条目的装置，所述条目包括到所述源表示库中的所述运行时条件代码的所述部分的第一引用、所述代码上下文和到从所述执行代码库中的所述运行时条件代码生成的所述条件机器代码的第二引用。

17.根据权利要求10所述的系统，其中，所述执行上下文还包括分配表，所述分配表包括针对所述程序代码中的所述变量和用户定义类型分配的元素。

18.根据权利要求10所述的系统，其中，所述运行时条件代码包括函数，并且所述代码上下文包括所述函数利用其被调用的至少一个包含、参数或命名空间。

动态语言的优化执行

技术领域

[0001] 本公开一般地涉及动态计算机程序语言和即时(just-in-time)代码生成领域。具体地,本公开描述了一种通过上下文跟踪和即时代码生成来优化执行动态计算机程序语言的系统、方法和过程。

背景技术

[0002] 针对web开发和应用开发已大量采用了诸如PHP:超文本预处理器(PHP)、Python、Ruby和Perl的动态语言,因为它们使得开发者产率高。这些语言有许多特征有助于它们的多产特性,但是最重要的两个特征是运行时键入和将新数据动态地评价为代码的能力。这些特性允许开发者编写能够比以诸如C、Java或C++的语言编写的静态代码更灵活地使用的代码。例如,动态语言通过在运行时允许开发者有条件地从现有代码生成新的可执行代码以处理新数据,来使得能够进行更灵活的开发实践。

[0003] 动态语言的缺点在于它们的执行通常显著慢于静态语言(其可能强制编译时静态键入并且可能不允许动态代码执行)。由于在编译时函数和变量的类型是已知的,并且由于可执行代码的全范围是编译器已知的,所以可在程序的执行之前有效地将静态语言转换成机器代码。在动态语言的情况下,在代码被执行的时刻之前,可执行代码的全范围未知,并且许多函数和变量的类型可能未知。由于这些原因,动态语言无法在执行之前被完全地转换为机器代码。另外,即使向机器代码的运行时转换常常也是不可行的,因为即时编译过程本身可能需要大量执行资源,导致执行期间的性能下降。

发明内容

[0004] 所公开的配置包括一种通过动态执行模块来执行计算机程序的系统、方法和计算机可读介质。所公开的配置包括一种计算机实现的方法,其中,接收执行程序代码的第一请求。这里,所述程序代码包括初始代码和运行时条件代码。响应于第一请求,从初始代码生成执行资源,其中所述执行资源包括执行代码库(execution codebase)和执行上下文(execution context)。所述执行代码库包括从初始代码生成的机器代码,并且所述执行上下文包括存储的对针对初始代码中的函数、变量和用户定义类型生成的元素的引用。执行代码库中的机器代码被执行,并且使用运行时条件代码来更新执行代码库和执行上下文。然后接收执行程序代码的第二请求,并且使用更新的执行上下文来执行更新的执行代码库中的机器代码。

附图说明

[0005] 图1是示出执行环境中的动态执行模块的示例实施例的高级框图。

[0006] 图2是示出示例计算机的高级框图。

[0007] 图3是示出根据一个实施例的动态执行模块内的组件的详细视图的框图。

[0008] 图4是示出使用动态执行模块的一个实施例来执行程序代码的过程的流程图。

具体实施方式

[0009] 附图和以下描述仅以例示方式描述特定实施例。本领域技术人员将容易从以下描述认识到在不脱离本文所述的原理的情况下可采用本文所示的结构和方法的替选实施例。现在将详细参照若干实施例，其示例示出于附图中。需要注意的是，只要可行，相似或相同的参考数字可用在图中并且可指示相似或相同的功能。

[0010] 图1示出在执行环境120中的程序代码100和动态执行模块101。执行环境120包括可托管并执行计算机程序指令的一个或多个计算机系统。在一个实施例中，执行环境120是单个计算机系统，例如web服务器。在另一实施例中，执行环境包括通过网络连接的若干算机系统，例如数据中心中的服务器计算机。

[0011] 例如，程序代码100包括以诸如Python、Perl、Ruby、PHP等的计算机语言的计算机源代码。程序代码100可包括初始代码106和运行时条件代码107。初始代码106包括在发起程序代码100的执行时被加载的计算机指令。例如，初始代码106可包括具有程序的“主函数”或“主循环”的文件。初始代码106在被执行时可加载调用运行时条件代码107的附加计算机程序指令。运行时条件代码107可基于在已发起执行之后确定的情况来有条件地加载。例如，运行时条件代码107可包括在运行时基于程序代码100的当前执行从数据库加载的计算机指令。

[0012] 动态执行模块101是执行环境120的组件，其使得程序代码100能够更有效地执行。动态执行模块101取代了用于诸如PHP、Perl、Python或Ruby的编程语言的标准虚拟机或解析器。动态执行模块101可以是在还执行执行环境120的其它进程的硬件上执行的软件模块，或者其可以在它自己的专用硬件资源上执行。当程序代码100在执行环境120中执行时，动态执行模块101管理并监视该执行以确保执行既快速还有效。动态执行模块101可通过使用即时机器代码生成、代码片段跟踪、机器代码重用、上下文跟踪以及其它技术来改善程序代码100的执行。本文中更详细地描述了这些技术中的一些。

[0013] 动态执行模块101包括执行资源102和执行支持模块105。执行资源102是为特定程序代码100生成的资源，其中一些资源可跨越相同代码的若干调用一直存在。执行资源102包括执行上下文103和执行代码库104。执行上下文103存储特定程序在执行时的动态运行时状态以及可跨越不同执行一直存在的程序代码100的结构和状态。执行上下文103包括结构，该结构存储对针对程序代码100中的函数、变量和用户定义类型(例如类、结构等)生成的元素的引用。执行上下文103可在程序代码100的调用之间被重置，以防止来自一个调用的动态值污染代码的下一调用。然而，执行上下文重置可被执行为使得保持非易失性数据(其可在调用之间迁移)。执行代码库104存储在程序的该执行以及先前执行期间从程序代码100生成的机器代码。执行代码库104用作从程序代码100生成的机器代码的缓存，使得在程序的调用之间不需要重复代码生成。执行上下文103用于跟踪执行代码库104中与程序的当前调用相关的机器代码部分(例如，用于当前调用的函数定义的机器代码等)。执行支持模块105为动态执行模块101提供运行时支持，例如即时机器代码生成、执行管理等。

[0014] 图1所示的实体利用一个或多个计算机或者这样的计算机内的模块来实现。图2是示出示例计算机200的高级框图。计算机200包括耦合到芯片集204的至少一个处理器202。芯片集204包括存储器控制器中心(hub)220和输入/输出(I/O)控制器中心222。存储器

(memory)206和图形适配器212耦合到存储器控制器中心220，显示器218耦合到图形适配器212。存储装置208、键盘210、指示装置214和网络适配器216耦合到I/O控制器中心222。计算机200的其它实施例具有不同的架构。

[0015] 存储装置208是诸如硬盘驱动器、压缩盘只读存储器(CD-ROM)、DVD或固态存储装置的非瞬时性计算机可读存储介质。存储器206保存处理器202所使用的指令和数据。指示装置214是鼠标、跟踪球或其它类型的指示装置，并且与键盘210组合使用以向计算机200输入数据。图形适配器212在显示器218上显示图像和其它信息。网络适配器216将计算机200耦合到一个或多个计算机网络。

[0016] 计算机200适于执行计算机程序模块以提供本文所述的功能。如本文所用，术语“模块”是指用于提供指定功能的计算机程序逻辑。因此，模块可实现于硬件、固件和/或软件中。在一个实施例中，程序模块被存储在存储装置208上，被加载到存储器206中，并由处理器202执行。

[0017] 图1的实体所使用的计算机200的类型可根据实施例以及实体所需的处理能力而变化。例如，执行环境120可以包括一起工作以提供本文所述的功能的多个刀片服务器。计算机200可缺少上述组件中的一些，例如键盘210、图形适配器212和显示器218。

[0018] 图3是示出根据一个实施例的动态执行模块101内的组件的详细视图的高级框图。如前所述，动态执行模块101包括执行资源102和执行支持模块105。执行资源102进一步包括执行上下文103和执行代码库104。

[0019] 如前所述，执行上下文103存储捕获程序执行的当前运行时状态的数据以及可跨越程序调用一直存在以加速同一程序代码的未来执行的信息。执行上下文103包括函数表301、变量表302、分配表304、源表示库309和包含表(include table)310。执行上下文103还可包括用于各种用户定义类型的表，例如类表303。用户定义类型所需的表将取决于动态执行模块101所执行的特定动态语言。在PHP的情况下，可使用类表303，但是在没有类的非面向对象的语言的情况下，类表303可没有必要，而可利用相似的表来跟踪其它用户定义类型。

[0020] 分配表304是数据结构，其包含初始代码106中定义的非易失性数据以及在运行时在程序的调用期间动态分配的元素。例如，分配表304存储程序代码100中声明的变量、结构和用户定义类型(例如类)。动态执行模块101可通过基于程序代码100的先前调用所使用的存储器在分配表304中预先分配空间并且通过在执行上下文103的重置之间保持分配表304中的非易失性数据，来减少程序代码100所进行的运行时存储器分配的量。跨越程序代码100的单独的执行一直存在的数据的示例包括全局常量、全局变量、全局类等。存储在分配表304中的元素可被变量表302和类表303中的条目引用。

[0021] 函数表301是按照名称和代码上下文将函数映射至执行代码库104中的编译的机器代码的数据结构。代码上下文是利用其调用函数的上下文，例如参数、包含、命名空间等。在动态语言中，函数调用的代码上下文可改变函数调用所执行的代码。例如，改变绑定至函数名称的代码的函数调用之前的条件包含语句可导致该函数调用与它在先前执行中所调用的不同的代码片段，即使函数名称和参数相同。

[0022] 当程序代码100被执行并且遇到对先前在相同代码上下文中(在程序代码100的先前执行中或者在早前时间的相同执行期间)定义的函数的调用时，该函数的机器代码生成

将没有必要，因为可通过函数表301中的映射来访问相同代码上下文中针对相同函数先前生成的机器代码。另一方面，如果遇到新函数，或者如果在新的代码上下文中遇到已知函数，则可针对该函数生成新机器代码，并且可将对该机器代码的引用插入函数表301中的适当条目中。这样，过去已产生的机器代码可被重用，而不会牺牲基于上下文调用不同代码的动态生成的函数的灵活性。

[0023] 变量表302将变量名称映射至分配表304中存储的变量。类似地，用户定义类型可具有将那些类型的命名实例映射至分配表304中的分配的表。例如，类表303将类名称映射至分配表304中存储的类。可在运行时基于变量或用户定义类型的改变代码上下文来修改该映射。这使得动态执行模块101能够支持运行时对变量和用户定义类型的动态改变，同时在变量和用户定义类型是非易失性的情况下仍能够优化和重用。

[0024] 函数表301、变量表302以及用于用户定义类型的表(例如类表303)可使得没有被动态修改的无条件内置数据能够跨越程序代码100的独立执行而一直存在。无条件内置数据的该持续性可通过使内置非易失性数据所需的初始化最小化来加速程序代码100的新调用所需的初始设置。

[0025] 源表示库309包括程序代码100中已由动态执行模块101处理的指令的表示。源表示库309可按照能够更快访问指令结构以便于动态执行模块101处理的方式来存储程序代码100的指令。例如，在一个实施例中，来自程序代码100的指令按照控制流图(CFG)的形式存储在源表示库309中。在另一实施例中，程序代码100的指令按照抽象语法树(AST)的形式存储。这样的结构实施例可加速代码生成和代码索引。

[0026] 例如，假设程序代码100包括如下PHP指令：

```
<?php
function __autoload($className) {
    class ExampleClass {
        var $exampleProp;
    }
}
$object = new ExampleClass();
```

[0027]

```
$object->exampleProp = $_GET['arg'];
if ($object->exampleProp) {
    function Conditional() { return "First Condition\n"; }
} else {
    function Conditional() { return "Second Condition\n"; }
}
print Conditional();
?>
```

[0028] 在一个实现方式中，源表示库309可包含对应CFG，其(被表示为文本，而非二进制数据)可看起来像：

```
CFG: <PHPEntryFunc>: 4 blocks, 9 registers, and 10 constants.
```

[0029]

```
Block[0], 11 op(s):
  DEFINE_FUNC(__autoload)
```

```
    %0 = VARIABLE($object ($0), #0)
    %1 = CLASS_CTOR("ExampleClass", %0)
    ASSIGN<,>(%0, %1)
    %2 = SUPERGLOBAL<2>
    %3 = ARRAY_ACCESS(%2, "arg")
    %4 = VARIABLE($object ($0), #1)
    %5 = PROP_ASSIGN<,>(%4, "exampleProp", %3)
    %6 = VARIABLE($object ($0), #2)
    %7 = PROP_REF<0>(%6, "exampleProp")
    BR<1>(%7, Block[1], Block[2])
  Block[1], 2 op(s):
    DEFINE_FUNC(Conditional [v1])
    BR<0>(Block[3])
  Block[2], 2 op(s):
    DEFINE_FUNC(Conditional [v2])
    BR<0>(Block[3])
[0030] Block[3], 3 op(s):
    %8 = CALL<1>("Conditional", %0)
    ECHO<1>(%8)
    RETURN

  CFG: __autoload(): 1 block, 0 registers, and 3 constants.
  Block[0], 2 op(s):
    DEFINE_CLASS<0>(ExampleClass)
    RETURN

  CFG: Conditional() [v1]: 1 block, 0 registers, and 3 constants.
  Block[0], 1 op(s):
    RETURN("First Condition\n")

  CFG: Conditional() [v2]: 1 block, 0 registers, and 3 constants.
  Block[0], 1 op(s):
    RETURN("Second Condition\n")
```

[0031] 以上是一个可能CFG的内容的格式化打印输出。在存储器中，此CFG可以不同的形式存储。CFG可根据动态执行模块的具体实现细节而以不同的形式结构化，并且在被打印时，对于相同PHP代码，这些CFG可给出不同于上面所示的文本表示。

[0032] 包含表310包含程序代码100中的指令至针对那些指令生成的可存储在执行代码库104中的机器代码的映射。当程序代码100被动态执行模块101处理时，程序代码中的指令的源表示被存储在源表示库309中。包含表310保存在运行时导入的每条代码的条目以及对该条代码的机器代码所位于的执行代码库104的引用。包含表310中的条目还可包括对源表示库309的引用，以指示与机器代码对应的程序代码指令。该条目可包括跟踪自生成机器代码起程序代码100是否已改变的机制(例如时间戳)。如果检测到改变，则动态执行模块101将激活进程以针对程序代码100的改变的部分生成新的机器代码。

[0033] 执行代码库104存储由动态执行模块101从程序代码100生成的机器代码。当给定程序代码100被首次执行时，执行代码库104不包含用于该程序的机器代码。当程序被首次执行时动态执行模块101从程序代码100生成机器代码。在同一程序代码100的后续执行时，执行代码库104将具有对于该程序而言已经存在的机器代码。该先前生成的机器代码可被直接执行，而无需进一步的代码生成。如果程序代码100的当前调用执行过去没有执行过的指令并且在执行代码库104中没有针对其的机器代码，则动态执行模块101将针对这些指令生成机器代码并且将它们存储在执行代码库104中。执行代码库104中的机器代码包括用户代码和系统代码。用户代码对应于直接从程序代码100中的指令生成的机器代码。系统代码是由动态执行模块101生成以将用户代码联系在一起并且使其能够随执行上下文102以及动态执行模块101的其它组件操作的代码。系统代码使得用户代码能够利用通过执行上下文103实现的重用性和性能优化。例如，对于程序代码100中的变量引用，动态执行模块101可生成机器代码，所述机器代码在变量表302中查找用于该变量的条目，然后使用该表条目作为程序中的变量值。

[0034] 例如，假设程序代码100包含以下PHP代码：

```
<?php
function ExampleFunction()
```

[0035]

```
{
    $var = "Example String\n";
    $varVar = 'var';

    print $$varVar;
}
```

[0036]

```
ExampleFunction();
?>
```

[0037] 可由动态执行模块101针对以上PHP代码生成以下机器代码：

[0038]

```
ExampleFunction():
    push    %rbp
    mov     %rsp,%rbp
    push    %r15
    push    %r12
    push    %rbx
```

```
    sub     $0x20,%rsp
    movabs $0x7ffff7feadd0,%rax
    callq   *%rax              ; DynVariables_Enter()
    mov      %rax,%r15
    movabs $0x7ffff01f2410,%rdi
    lea     -0x40(%rbp),%rsi
    movabs $0x7ffff7feae40,%rax
    movabs "var",%rcx
    movabs "Example String\n",%rdx
    movl    $0x4,-0x34(%rbp)
    movl    $0x4,-0x24(%rbp)
    mov      %rdx,-0x40(%rbp)
    movl    $0x2,-0x34(%rbp)
    mov      %rcx,-0x30(%rbp)
    movl    $0x2,-0x24(%rbp)
    mov      %r15,%rdx
    callq   *%rax              ; DynVariables_Lookup()
    cmpl    $0x2,0xc(%rax)
    jne     b0
    mov      (%rax),%rax
    jmpq    b1
b0:
    movabs "<Current file name>",%rsi
    movabs $0x7ffff7f51f88,%rcx
    mov      %rax,%rdi
    callq   *%rcx              ; ConvertToString()
b1:
    mov      %rax,%r12
    movabs $0x7ffff61d7be0,%rax
    mov      %r12,%rdi
    callq   *%rax              ; strlen()
    movabs $0x7ffff7f51c40,%rcx
    mov      %r12,%rdi
    mov      %rax,%rsi
    callq   *%rcx              ; DoOutput()
    movabs $0x7ffff04c2aa0,%rax
    mov      %rbx,(%rax)
    movabs $0x7ffff01f2410,%rdi
    movabs $0x7ffff7feadf0,%rax
    mov      %r15,%rsi
    callq   *%rax              ; DynVariables_Reset()
    movabs thePHPNull,%rax
    add     $0x20,%rsp
    pop      %rbx
    pop      %r12
    pop      %r15
    pop      %rbp
    retq
```

[0039]

[0040] 这里，DynVariables函数是来自运行时工具(runtime utility)311的更新并访问变量表302的函数。ConvertToStrtng函数将任意类型的变量转变为字符串，而strlen函数

是给出字符串的长度的标准C库函数。DoOutput函数是来自运行时工具311的生成输出的函数。该机器代码通过使用来自运行时工具311的这些函数与执行资源102(例如,变量表302)对接,执行与PHP代码相同的过程。这些函数向程序提供对用于所有函数、变量、类等的动态值的访问,从而使得机器代码程序能够以动态语言来实现程序。

[0041] 执行支持模块105是动态执行模块101的组件,其为动态执行过程提供支持和管理。执行支持模块105包括代码生成器307、执行管理器308和运行时工具311。执行管理器308发起、监视、管理和终止动态执行模块101中的程序代码100的执行。当使用动态执行模块101调用程序代码100时,执行管理器308根据需要协调动态执行模块的其它组件的激活,使得程序代码100被正确地执行。

[0042] 执行管理器308激活代码生成器307以从程序代码指令生成机器代码。生成的机器代码被存储在执行代码库104中,使得相同机器代码中的至少一些可在程序代码100的不同调用中被重用。这是一种优化,有助于使针对程序代码100进行的代码生成最小化。可跨越程序代码100的分立调用以及在程序代码的同一调用内相同代码片段被执行多次的情况下,进行该优化。执行管理器308通过使用包含表310和源表示库309跟踪程序代码100的已经在执行代码库104中存储有对应机器代码的部分,来确定给定程序代码100所需的代码生成。当使用代码生成器307来生成机器代码时,对应程序代码片段被放置在源表示库309中,并且具有对此代码片段的引用的条目随对执行代码库104中的机器代码的引用和该片段的代码上下文一起被放置在包含表310中。如果在执行期间再次遇到相同代码上下文中的相同代码片段,则执行管理器308将使用执行代码库104中的现有机器代码。如果包含表310不包含给定代码片段的条目(或者如果代码片段具有不同的代码上下文),则执行管理器308将激活代码生成器307以针对该代码片段在其当前代码上下文中生成机器代码。生成的机器代码将被存储在执行代码库104中,具有对代码片段的引用的条目、其当前代码上下文和对生成的机器代码的引用将被存储在包含表310中。

[0043] 此方案的效果在于它使得能够在过去没有遇到过的组合中重用机器代码。例如,假设PHP程序包含名为Foo()和Bar()的函数,这两个函数分别具有两个不同的声明:Foo(v1)、Foo(v2)和Bar(v1)、Bar(v2)。还假设此PHP程序运行两次,在第一次运行时执行Foo(v1)和Bar(v1),在第二次运行时执行Foo(v2)和Bar(v2)。如果在第三次运行时需要执行Foo(v1)和Bar(v2),则将不需要机器代码生成,因为执行代码库104将具有用于Foo(v1)和Bar(v2)二者的机器代码,尽管函数执行的这种特定组合之前没有看到过。

[0044] 执行管理器308还负责在执行环境120中协调相同程序代码100的多个实例的执行。如果执行环境120允许相同程序代码100的多个实例同时执行,则必须管理动态执行模块101的资源以防止冲突。例如,执行相同程序代码100的两个进程可能同时尝试修改执行上下文103。执行管理器308协调这两个进程以使得相同的公共资源的这些修改可在没有死锁、竞态条件或其它多处理错误的情况下进行。执行管理器308可使用各种多进程管理技术来防止这样的错误。这些多进程管理技术是本领域已知的,在本公开的范围之外。在一个实施例中,执行管理器308保持有执行资源102的主副本,并且针对各个执行程序代码实例生成资源的单独副本。在此实施例中,在程序代码实例的执行终止之后对执行资源102的改变被写回到主副本。

[0045] 运行时工具311是辅助执行管理器308提供用于动态执行进程的基础结构的模块。

运行时工具311包括可从程序代码100生成CFG或AST结构的进程、可在程序代码的执行终止之后重置执行上下文103的进程以及管理函数表301、变量表302、用于用户定义类型的表、分配表304和包含表310的进程。

[0046] 图4是示出使用动态执行模块的一个实施例执行程序代码的过程的流程图。当动态执行模块101接收到401执行程序代码100的请求时所述过程开始。所述请求可接收自执行环境120中的另一模块。例如，web服务器进程可请求为用户所请求的网页执行PHP程序代码。

[0047] 然后，执行管理器308将确定402是否存在用于该程序代码的执行资源102。如果动态执行模块在过去执行过相同的程序代码100，则在动态执行模块101中将存在执行资源102。如果执行资源不存在403，则执行管理器308将激活运行时工具311来生成405新的执行资源102(即，执行上下文103和执行代码库104)。执行上下文103的组件(即函数表301、变量表302、用户定义类型表(类表303)、分配表304、源表示库309和包含表310)将被扩增基于初始代码106的处理的数据。初始代码将被解析，并且其中的指令的表示将被存储在源表示库309中。从这些初始指令生成的机器代码将被存储在执行代码库104中，包含表310将被扩增将源表示库中的代码片段与执行代码库中的对应机器代码联系起来的条目。随着初始代码106被处理以生成源表示和机器代码，用于遇到的函数、变量和用户定义类型的条目分别被存储在函数表301、变量表302和用户定义类型表中。这些元素的分配被存储在分配表304中。

[0048] 如果用于程序代码100的执行资源已经存在403，则现有的执行上下文(已从先前调用重置)将被初始化410。执行上下文103的初始化可涉及：基于由执行管理器308从相同程序代码的先前执行估计的要求，为执行上下文预分配资源。例如，可基于先前执行的分配大小为分配表304分配存储器。另外，用于给定程序代码100的执行资源102的组件可预先扩增有基于相同程序代码的先前执行的数据。例如，如果在动态执行模块101中已执行过相同程序代码，则执行代码库104将包含已从先前执行中处理的程序指令生成的机器代码。如果先前已执行过程序代码，则很可能此程序代码的初始代码106已被处理过，并且可在执行代码库104中找到用于此初始代码的机器代码。类似地，源表示库309和包含表310可包括先前处理的程序指令的源表示和包含表条目。尽管来自程序代码100的先前执行的所有函数数据、变量数据和用户定义类型数据不适用于相同程序代码的后续执行，但是该数据中的一些可被重用。该数据可在执行之间被预先初始化(或保留)在函数表301、变量表302和各种用户定义类型表中。例如，初始代码106中声明的函数、变量和类可分别被保留在函数表301、变量表302和类表303中，因为初始代码106中声明的元素可能适用于给定程序代码100的任何调用。

[0049] 一旦执行资源102已被初始化，程序执行管理器308就通过动态执行模块101执行415程序代码100。程序代码100的执行是不同于通过执行环境120中的其它手段执行程序代码100的进程。在没有动态执行模块的情况下程序代码的典型执行涉及使用解释器来处理程序代码的指令，以生成可在虚拟机上执行的字节代码。在动态执行模块101的情况下，通过执行与初始代码106对应的执行代码库104中的机器代码指令来发起程序代码100的执行。无论执行资源102是从先前数据初始化的410还是重新生成的405，这些机器代码指令存在于执行代码库104中。执行代码库104中的机器代码的执行所生成的结果在功能上等同于

通过执行环境中的常规进程来执行对应程序代码指令。尽管机器代码不直接拥有诸如PHP、Python、Ruby、Perl等语言的动态方面，但是动态执行模块101通过执行上下文103和执行支持模块105来实现该动态机制。例如，随着与程序代码100对应的机器代码执行，它可利用基于代码上下文引用不同元素的函数、变量或用户定义类型。这实现了动态语言的动态声明和绑定。如果通过直接调用函数、变量和用户定义类型来整体地生成执行代码库104中的机器代码，则这种动态机制将是不可能的。然而，由于对于动态实体，执行代码库104中的机器代码没有直接引用函数、变量和用户定义类型，而是引用函数表、变量表和用户定义类型表中的条目，这些条目提供一个间接层，使得动态实体的执行代码所使用的元素可基于代码上下文而改变。

[0050] 随着执行代码库104中的机器代码执行，它可尝试从运行时条件代码107加载附加代码片段。如果动态执行模块101先前已处理过此代码片段(在程序代码100的此调用中或者在先前调用中)，则将在源表示库309中找到此代码片段的表示，并且包含表310将具有用于该片段的条目以及其先前执行的代码上下文。如果是在相同的代码上下文中调用该代码片段，则将在执行代码库104中找到与该代码片段对应的机器代码。如果运行时条件代码107中的代码片段自从其生成此机器代码起没有改变过，则可直接执行该机器代码。然而，如果遇到416新的代码片段或代码上下文(或者如果与代码片段对应的运行时条件代码107自针对其生成机器代码起改变过)，则执行管理器308将激活代码生成器307以生成与代码片段对应的机器代码指令。这些机器代码指令将被存储在执行代码库104中，并且将利用对执行代码库中的机器代码的引用、代码片段在源表示库309中的源表示以及对代码上下文的引用，在包含表310中创建对应条目。

[0051] 在执行代码库104中生成新的机器代码以及执行现有的机器代码可均导致执行资源102需要更新418。可通过执行管理器和/或一个或多个运行时工具311来更新419执行资源102。执行资源更新可包括在函数表301、变量表302、用户定义类型表(例如，类表303)、分配表304和包含表310中添加或删除条目。更新还可包括从执行代码库104添加或移除机器代码以及从源表示库309添加或移除代码片段的源表示。例如，当通过动态执行模块101处理新代码片段时，将分别针对片段中发现的函数、变量和类生成函数表301、变量表302和类表303中的新条目。从新代码片段生成的机器代码将被存储在执行代码库104中，并且源表示库309将被更新以包括该片段的表示。函数、变量和用户定义类型条目还可被更新以包括对存储在执行代码库104中的新机器代码的引用。然后程序代码100的执行415继续。

[0052] 程序代码100的执行可最终终止。可由于程序指令本身到达结束点并离开程序，或者程序可能由于其它原因而被停止，而发生程序的终止。例如，由于任务已完成或者为了释放系统资源，可通过执行环境120来终止程序。如果执行终止420，则执行管理器308将使得执行资源102被重置421。重置执行资源102可涉及从函数表301、变量表302、用户定义类型表和分配表304删除条目。执行管理器308将尝试跨越程序代码100的调用保留这些表中的尽可能多的条目；然而，一些条目跨越不同的调用将不适用，因为它们的值将易于根据程序代码中执行的指令而变化。例如，用于变量存储302中的变量的条目将几乎总是被删除，因为变量的值将取决于导致它们的指派的操作的顺序。被全局指派一次的变量(例如，常量)可以是能够跨调用保留的例外情况。另一方面，执行管理器308可将存储在源表示库309中的源表示、执行代码库104中的机器代码以及包含表310中的条目全部保留。源表示库309、

包含表310和执行代码库104一起提供避免针对之前已处理过一次的代码片段再次生成机器代码的手段。然而，如果程序代码100剧烈地改变，或者如果在任何执行资源102中发现错误，则执行管理器308可通过完全释放执行资源102(删除资源的所有组件)来将整个系统硬重置。执行硬重置删除了动态执行模块101关于给定程序代码100所保留的所有信息，程序代码100的下一执行将就像它是该代码的首次执行一样进行(即，它将无法受益于任何先前代码生成)。

[0053] 为了示出上述过程，现在描述一个具体示例。假设程序代码100包含如下PHP代码：

```
<?php
function __autoload($className) {
    class ExampleClass {
        var $exampleProp;
    }
}
$object = new ExampleClass();
$object->exampleProp = $_GET['arg'];
if ($object->exampleProp) {
    function Conditional() { return "First Condition\n"; }
} else {
    function Conditional() { return "Second Condition\n"; }
}
print Conditional();
?>
```

[0054]

[0055] 动态执行模块的初始状态将如下(以人可读文本来表示，而非实际二进制表示)：

[0056] 类表：

[0057] Name->ID(Table)

[0058] "stdClass"->0

[0059] "ExampleClass"->8

[0060] ID->Active Definition(Array)

[0061] [0]Prototype for stdClass

[0062] [8]<empty>

[0063] 函数表：

[0064] Name->ID(Table)

[0065] "_autoload"->0

[0066] "Conditional"->1

[0067] ID->Active Target(Array)

[0068] [0]<empty>

[0069] [1]<empty>

[0070] 源表示库(CFG)：

[0071] "Main"/Top-level CFGs

[0072] -"PHPEntryFunc"("main")

[0073] Function->CFG(Table)

[0074] Conditional(v1)->CFG for Conditional("First Condition")

[0075] Conditional(v2)->CFG for Conditional("Second Condition")

[0076] 执行代码库：

[0077] -"PHPEntryFunc"

[0078] 在此示例中，基于ExampleClass的执行，动态执行模块101的执行资源102已被初始化，但是还未生成机器代码和状态。上面的“类表”部分示出基于类表303中的条目输出的文本。这里仅stdClass具有有效定义。“函数表”部分示出基于函数表301中的条目输出的文本。这里我们看到表中存在用于_autoload函数和Conditional()函数的两个条目。这些函数的“Active Target”部分还未填充，因为PHP还未执行过，在执行代码库中针对这些函数还没有机器代码。“源表示库”部分示出基于存储在源表示库309中的CFG的状态输出的文本。这里我们有用于PHPEntryFunc以及用于两个Conditional()函数声明的CFG。PHPEntryFunc是默认函数，其针对动态执行模块101中运行的每一个PHP程序而初始化并执行。动态执行模块101的一些实现方式可能没有这样的函数，或者可能具有不同的发起方案。“执行代码库”部分示出基于执行代码库104的内容输出的文本。这里，执行代码库104仅包含用于PHPEntryFunc的机器代码，这是开始程序的执行所需的最少量的机器代码。还未针对此PHP程序生成其它机器代码。

[0079] 现在，如果上面所示的PHP程序以arg＝0作为参数来执行，则执行资源将被更新为如下所示。

[0080] 类表：

[0081] Name->ID(Table)

[0082] "stdClass"->0

[0083] "ExampleClass"->8

[0084] ID->Active Definition(Array)

[0085] [0]Prototype for stdClass

[0086] [8]Prototype for ExampleClass

[0087] 函数表：

[0088] Name->ID(Table)

[0089] "_autoload"->0

[0090] "Conditional"->1

[0091] ID->Active Target(Array)

[0092] [0]Pointer to machine code for_autoload

[0093] [1]Pointer to machinc code for Conditional(v2)

[0094] 源表示库(CFG)：

[0095] "Main"/Top-level CFGs

[0096] -"PHPEntryFunc"("main")

[0097] Function->CFG(Table)

[0098] Conditional(v1)->CFG for Conditional("First Condition")

[0099] Conditional(v2)->CFG for Conditional("Second Condition")

[0100] 执行代码库：

[0101] -"PHPEntryFunc"

[0102] -_autoload

[0103] -Conditional(v2)

[0104] 一旦程序已被执行，对执行上下文就存在若干改变。在类表303中现在除了stdClass以外还有用于ExampleClass的原型。在函数表301中现在有用于_autoload和Conditional(v2)函数的指向执行代码库104的指针。后一指针是以下事实的结果:arg＝0导致程序通过if-then-else语句的“else”分支执行，而非“if-then”分支。源表示库309没有改变，因为没有另外的PHP被引入系统中。另一方面，执行代码库104现在包含已针对_autoload和Conditional(v2)函数生成的机器代码。

[0105] 现在，如果上面所示的PHP程序以arg＝1作为参数来执行，则执行资源将被更新为如下所示：

[0106] 类表：

[0107] Name->ID(Table)

[0108] "stdClass"->0

[0109] "ExampleClass"->8

[0110] ID->Active Definition(Array)

[0111] [0]Prototype for stdClass

[0112] [8]Prototype for ExampleClass

[0113] 函数表：

[0114] Name->ID(Table)

[0115] "_autoload"->0

[0116] "Conditional"->1

[0117] ID->Active Target(Array)

[0118] [0]Pointer to machine code for_autoload

[0119] [1]Pointer to machine code for Conditional(v1)

[0120] 源表示库(CFG)：

[0121] "Main"/Top-level CFGs

[0122] -"PHPEntryFunc"("main")

[0123] Function->CFG(Table)

[0124] Conditional(v1)->CFG for Conditional("First Condition")

[0125] Conditional(v2)->CFG for Conditional("Second Condition")

[0126] 执行代码库：

[0127] -"PHPEntryFunc"

[0128] -_autoload

[0129] -Conditional(v2)

[0130] -Conditional(v1)

[0131] 这里在两个重要方面状态改变。首先，函数表301现在指向用于Conditional(v1)的机器代码，因为if-then-else语句的“if-then”分支被遍历。其次，执行代码库104现在包含用于Conditional(v1)函数的附加机器代码。需注意的是，Conditional(v2)也存在于执

行代码库中。如果PHP源代码的未来执行再次需要代码的该分支，则将不需要机器代码生成；仅函数表301中的指针将需要更新。

[0132] 以上描述的一些部分在算法过程或操作方面描述实施例。这些算法描述和表示常常被数据处理领域的技术人员用来向本领域其他技术人员有效传达他们的工作的实质。这些操作尽管在功能、计算或者逻辑上进行描述，但是应理解为通过计算机程序、微代码等来实现，所述计算机程序包括用于由处理器或等效电路执行的指令。另外，也已经证实将功能操作的这些布置表示为模块有时便利而不失一般性。可以用软件、固件、硬件或其任何组合来实现描述的操作及其关联的模块。

[0133] 如本文所用，对“一个实施例”或“实施例”的任何引用意味着结合该实施例描述的特定元件、特征、结构或者特性包括在至少一个实施例中。说明书中出现于各处的短语“在一个实施例中”不必都指代相同的实施例。

[0134] 一些实施例可使用表达“耦合”和“连接”及其派生词来描述。应当理解，这些术语并非意在作为彼此的同义词。例如，一些实施例可使用术语“连接”来描述以指示两个或更多个元件彼此直接物理或者电接触。在另一示例中，一些实施例可使用术语“耦合”来描述以指示两个或更多个元件直接物理或者电接触。然而，术语“耦合”也可意指两个或更多个元件不彼此直接接触，但是仍然彼此协作或者交互。实施例不限于这样的上下文。

[0135] 如本文所用，术语“包括”、“包含”、“具有”或者其任何其它变体旨在覆盖非排他含义的包括。例如，包括元素列表的过程、方法、制品或设备不必仅限于那些元素，而是可包括未明确列举或者这样的过程、方法、制品或设备所固有的其它元素。另外，除非相反地明确指出，否则“或者”指代包括含义的或者，而非排他含义的或者。例如，以下各项中的任一项满足条件A或者B：A为真(或者存在)并且B为假(或者不存在)、A为假(或者不存在)并且B为真(或者存在)以及A和B均为真(或者存在)。

[0136] 此外，本文采用“一个”或“一”来描述实施例的元件和组件。这样做仅是为了方便并且给出本公开的一般意义。应当理解，此描述包括一个或者至少一个，并且除非明显另有含义，否则单数也包括复数。

[0137] 在阅读本公开时，本领域技术人员将理解用于动态语言的优化执行的系统和过程的附加替选结构和功能设计。因此，尽管已经示出并描述了特定实施例和应用，但是将理解所描述的主题不限于本文所公开的精确构造和组件，可在本文所公开的方法和设备的布置、操作和细节方面进行本领域技术人员将清楚的各种修改、改变和变化。

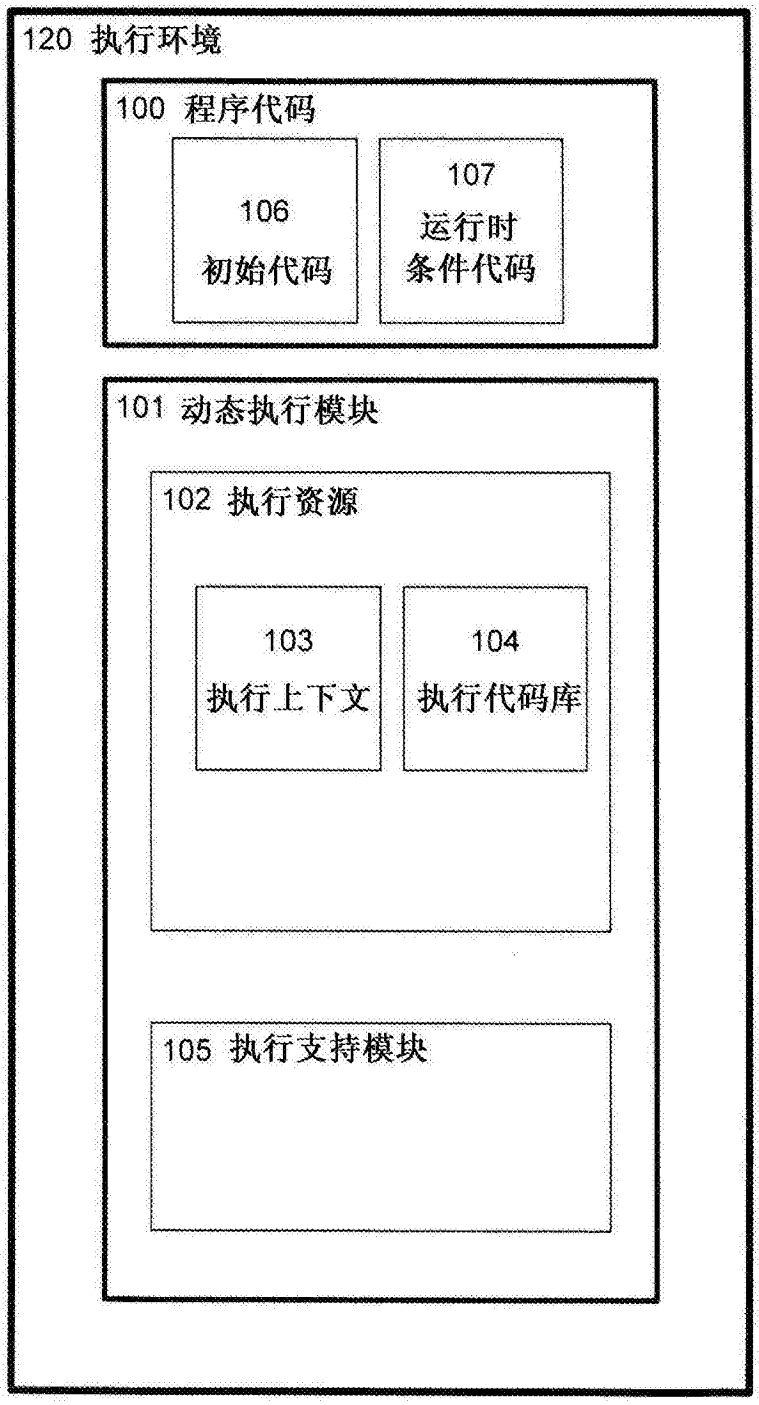

120 执行环境
100 程序代码
106
初始代码
107
运行时
条件代码
101 动态执行模块
102 执行资源
103
执行上下文
104
执行代码库
105 执行支持模块

图1

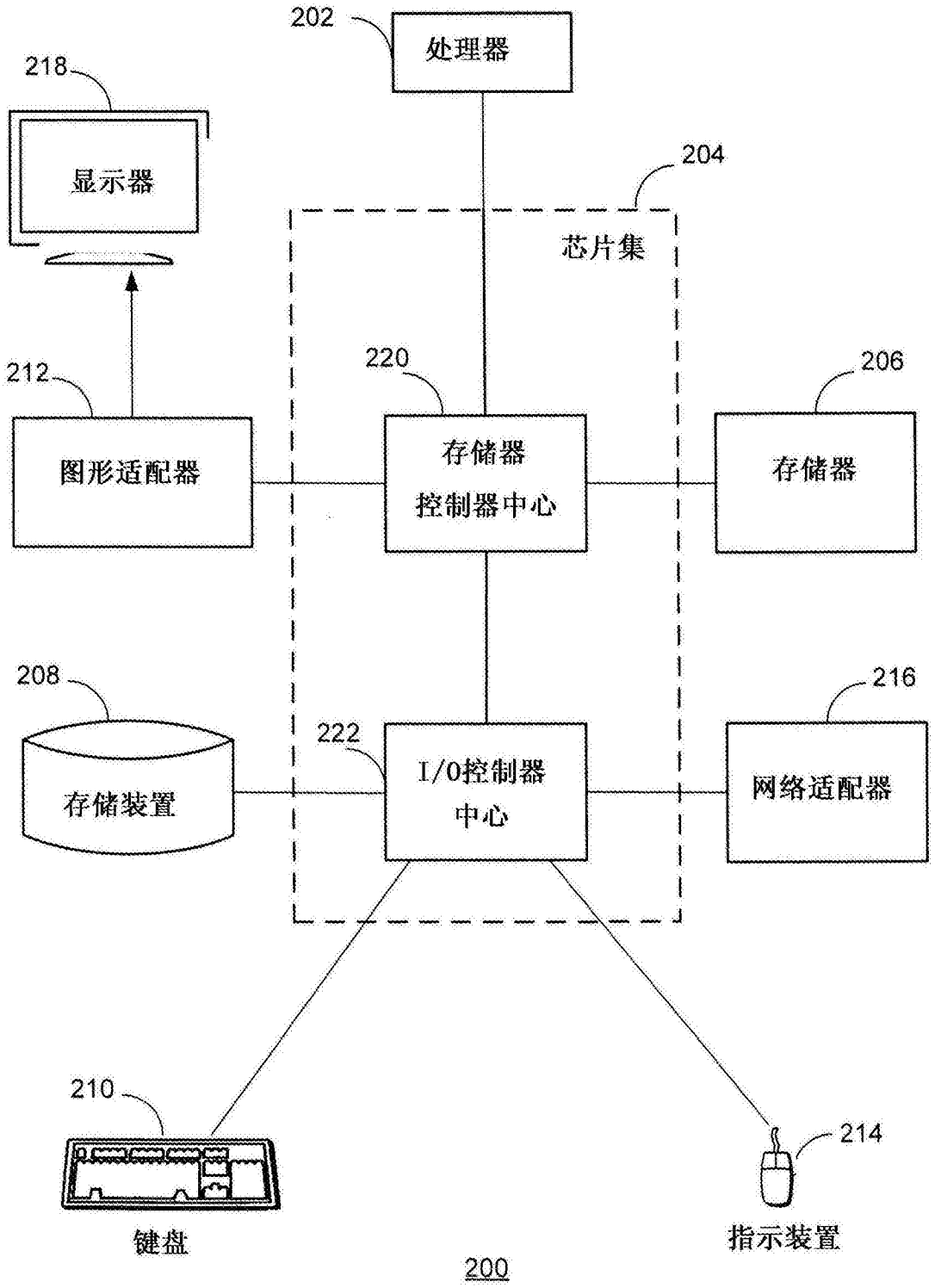

202
处理器
218
显示器
204
芯片集
212
220
206
图形适配器
存储器
控制器中心
存储器
208
216
222
存储装置
I/O控制器
中心
网络适配器
210
214
键盘
指示装置
200

图2

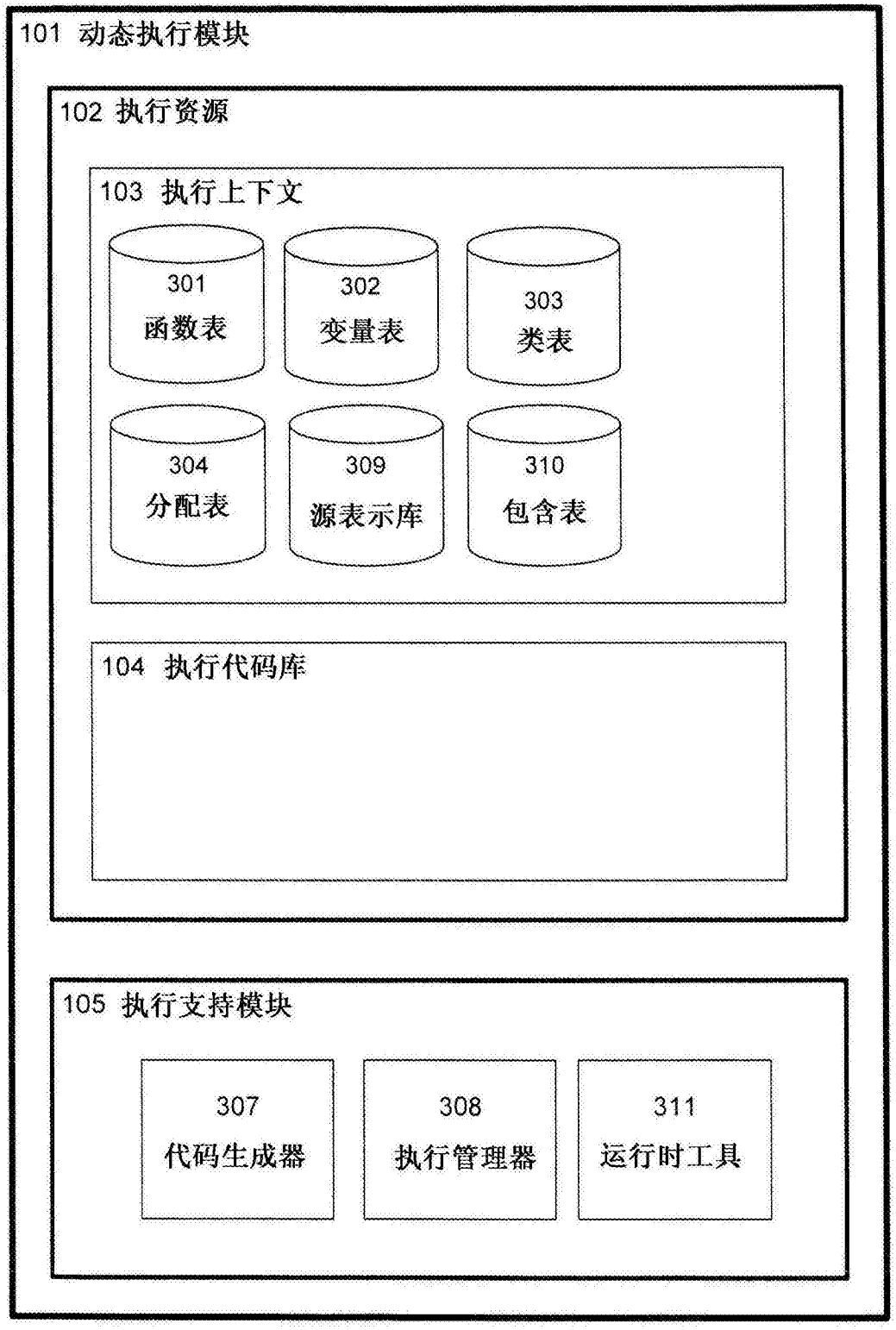

101 动态执行模块
102 执行资源
103 执行上下文
301
函数表
302
变量表
303
类表
304
分配表
309
源表示库
310
包含表
104 执行代码库
105 执行支持模块
307
代码生成器
308
执行管理器
311
运行时工具

图3

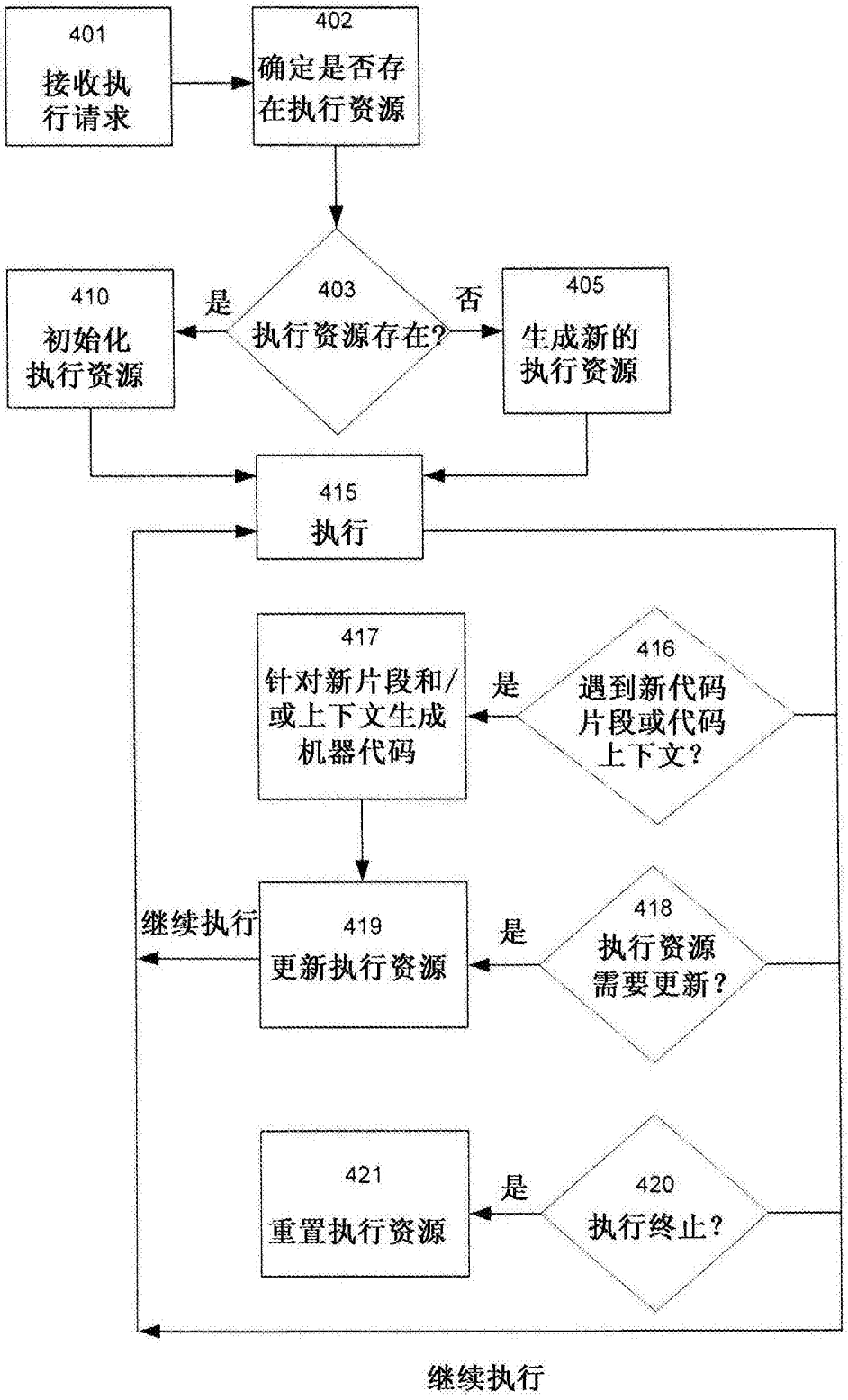

401
接收执行请求
402
确定是否存在执行资源
403
执行资源存在?
是
否
410
初始化执行资源
405
生成新的执行资源
415
执行
416
遇到新代码片段或代码上下文?
是
417
针对新片段和/或上下文生成机器代码
418
执行资源需要更新?
是
419
更新执行资源
继续执行
420
执行终止?
是
421
重置执行资源
继续执行

图4