(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 104115120 A
(43)申请公布日 2014.10.22

(21)申请号 201380008460.2
(22)申请日 2013.01.31
(30)优先权数据
13/368,330 2012.02.07 US
(85)PCT国际申请进入国家阶段日
2014.08.07
(86)PCT国际申请的申请数据
PCT/US2013/023958 2013.01.31
(87)PCT国际申请的公布数据
WO2013/119441 EN 2013.08.15
(71)申请人 微软公司
地址 美国华盛顿州
(72)发明人 L·拉弗里尼尔 C·曼
(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 9/45(2006.01)
G06F 9/30(2006.01)

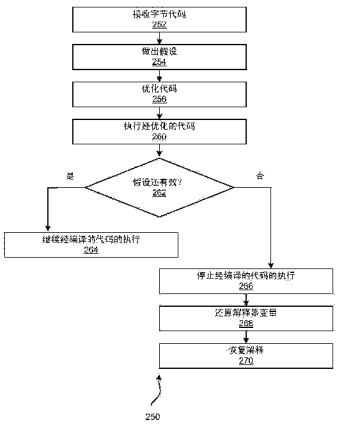
权利要求书2页 说明书14页 附图9页

(54)发明名称

将程序执行从编译代码变换到解释代码

(57)摘要

可以从源自以动态语言(例如JavaScript)编写的程序的字节代码来生成经优化的编译的代码。优化可以基于一种或多种假设。在经优化的编译的代码的执行期间,所述优化所基于的所述一种或多种假设可以被检查有效性。响应于确定所述优化是基于无效的一种或多种假设,经优化的代码的执行可以停止,解释器所使用的变量的状态可以被还原并且程序的执行可以在解释器中使用源自程序的未经优化的字节代码来恢复。程序的继续可以在字节代码中的一个点处恢复,所述点类似于在经优化的编译的代码中的所述一种或多种假设被确定为假的那点。



CN 104115120 A

1. 一种系统，包括：

计算设备的至少一个处理器；

所述计算设备的存储器；以及

包括加载到所述存储器中的至少一个模块的跳出引擎，所述至少一个模块使所述至少一个处理器：

响应于检测到关于程序在其上操作的数据的特性的至少一种假设的无效性，将以动态语言编写的程序的执行从机器代码执行器执行的经编译的代码变换到由解释器解释的经解释的字节代码，所述经编译的代码基于所述至少一种假设被优化，所述经解释的字节代码没有基于所述至少一种假设被优化；

在所述经解释的字节代码中的一个点处恢复在解释器中的所述程序的执行，所述点对应于在经优化的编译的代码中检测到无效性处的点。

2. 如权利要求 1 所述的系统，其特征在于，还包括：

加载到所述存储器中的至少一个模块，所述至少一个模块使所述至少一个处理器：

从由所述机器代码执行器所使用的对应变量中重新物化由所述解释器所使用的变量。

3. 如权利要求 1 所述的系统，其特征在于，还包括：

加载到所述存储器中的至少一个模块，所述至少一个模块使所述至少一个处理器：

在所述经优化的编译的字节代码中插入至少一个跳出点，所述至少一个跳出点与持有所述解释器恢复所述程序的执行所需要的值的一组变量相关联。

4. 如权利要求 1 所述的系统，其特征在于，其中执行是在经优化的编译的代码中预先确定的跳出点处被从经优化的编译的代码变换到未经优化的字节代码。

5. 一种方法，包括：

在计算设备的处理器处接收字节代码，所接收的字节代码包括源自以动态语言编写的程序的未经优化的字节代码；

接收关于由所述程序操作的数据的至少一种假设；

基于所述至少一种假设生成经优化的编译的代码；

执行所述经优化的编译的代码；

响应于确定所述至少一种假设的无效性，停止所述经优化的编译的代码的执行；

还原与所述经优化的编译的代码相关联的变量相对应的解释器变量；

在所述未经优化的字节代码中的一个点处恢复在解释器中的所述程序的执行，所述点对应于在经优化的编译的代码中确定所述无效性处的点。

6. 如权利要求 5 所述的方法，其特征在于，还包括：

捕捉在所述未经优化的字节代码中与在所述经优化的编译的代码中的至少一个预先确定的跳出点相对应的位置。

7. 如权利要求 6 所述的方法，其特征在于，还包括：

捕捉与在所述经优化的编译的代码中的至少一个预先确定的跳出点相关联的变量的位置。

8. 如权利要求 7 所述的方法，其特征在于，还包括：

执行所述经优化的编译的代码；以及

在所述至少一个预先确定的跳出点处重新物化与所述经优化的编译的代码变量相关

联的对应的解释器变量。

9. 如权利要求 5 所述的方法，其特征在于，还包括：

复活由所述解释器所使用的变量。

10. 一种包括当执行时使计算设备的至少一个处理器执行下列操作的计算机可执行指令的计算机可读存储介质：

响应于在跳出点处检测到至少一个无效情况，从执行源自以动态语言编写的程序的经优化的编译的代码中跳出，所述经优化的编译的代码基于至少一种无效假设被优化；

基于所述经优化的编译的代码变量的状态来还原解释器变量的状态；

在未经优化的字节代码中与在所述经优化的编译的代码中的所述跳出点相对应的点处，在所述解释器中恢复所述程序的执行；

执行源自所述程序的经优化的编译的代码，所述程序以 JavaScript 编写；以及

保存包括在所述经优化的编译的程序执行期间在其处存储了变量的位置的跳出信息。

将程序执行从编译代码变换到解释代码

[0001] 背景

[0002] 动态编程语言是一种在运行时执行静态语言通常在编译期间执行或根本不执行的行为的语言。在运行时由动态语言执行的行为可以包括通过添加新代码、通过扩展对象和定义或通过修改类型系统对程序的扩展。在动态编程语言中，在编译时不可用的信息可以显著地改变程序如何执行。例如，在静态程序中，当在源代码中声明变量时，该声明指定了所述变量的类型。变量 x 是整数或变量 x 是字符串等。如果变量 x 是整数，则将 x 加到整数常量将执行算术加法。如果变量 x 是字符串，则将 x 加到是字符串的常量将执行常量和 x 的串联。在动态编程语言中，变量的类型直到该程序执行时才知道，因此，需要额外的代码路径来处理不同类型的操作，这会出现为程序执行增加开销。

[0003] 通常被认为是动态语言的语言示例包括但不局限于：ActionScript、BASIC、BeanShell、ColdFusion、Common Lisp 和一些其它 Lisp 语言、Groovy、E 编程语言；JavaScript、VBScript、MATLAB、Lua、Objective-C、Perl、PHP、Powershell、Python、Ruby、Smalltalk、Tcl 以及 Dolphin Smalltalk。

[0004] 概述

[0005] 通过动态语言，可以做出关于不太可能静态确定的类型信息的假设。随后可以使用这种不精确的信息来应用优化。可以提供通过优化假设的无效所触发的后退或跳出机制。可以从源自以动态语言（例如但不局限于 JavaScript）编写的程序的字节代码来生成经优化的编译代码。优化可以基于关于数据的特性的一种或多种假设。在经优化的编译的代码的执行期间，所述优化所基于的所述一种或多种假设可以被检查有效性。响应于确定所述优化是基于无效的一种或多种假设，经优化的代码的执行可以停止，解释器所使用的对应的变量的状态可以被还原并且程序的执行可以在解释器中使用源自程序的未经优化的字节代码来恢复。程序的继续可以在字节代码中的一个点处恢复，所述点类似于在经优化的编译的代码中的所述一种或多种假设被确定为假的那点。

[0006] 在经优化的编译的代码中的优化可以包括对变量类型操作的优化。可以将传递给解释器的跳出信息量最小化。非活动的编译的代码不需要被复活用于解释器。

[0007] 可以在经优化的编译的代码中预先确定的点处设定一个或多个跳出点。在每个跳出点处，可以捕捉未经优化的字节代码中的变量位置和对应的位置。当在执行期间到达跳出点时，可以检查与该跳出点相关联的一种或多种假设的有效性，并且变量状态可以被还原用于对应的解释器的变量。可以将变量的状态和在未经优化的字节代码中的对应位置发送给解释器。解释器可以在未经优化的字节代码中的所接收的对应位置处恢复未经优化的字节代码的执行。

[0008] 提供本概述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概念选择。本概述并不旨在标识所要求保护主题的关键特征或必要特征，也不旨在用于限制所要求保护主题的范围。

附图说明

[0009] 在附图中：

[0010] 图 1a 示出根据在此所述的主题的各方面的包括被配置为呈现来自所接收的文档中的页的浏览器应用的系统 100 的示例；

[0011] 图 1b 示出根据在此所述的主题的各方面的包括被配置为从编译的代码变换到字节代码的执行的运行时引擎的系统 200 的示例；

[0012] 图 1c 示出根据在此所述的主题的各方面的包括被配置为从编译的代码变换到字节代码的执行的运行时引擎的系统 300 的示例；

[0013] 图 1d 示出根据在此所述的主题的各方面的程序源代码、对应的字节代码、编译的代码和跳出表格的示例 150；

[0014] 图 1e 示出根据在此所述的主题的各方面的具有跳出信息的编译的代码的示例 170；

[0015] 图 1f 示出根据在此所述的主题的各方面的程序源代码、以及具有跳出信息的编译的代码的示例 171；

[0016] 图 1g 示出根据在此所述的主题的各方面的程序源代码、对应的字节代码、以及具有跳出信息的编译的代码的示例 180；

[0017] 图 2 示出根据在此所述的主题的各方面的将程序执行从经优化的编译的代码变换到解释的代码的方法 250 的示例；

[0018] 图 3 是根据本文公开的主题的各方面的计算环境的示例的框图。

[0019] 详细描述

[0020] 概览

[0021] 一些基于静态分析的传统的编译器优化对于动态语言是不可用的，因为进行优化所需的信息在编译时不可用。然而，通过使用试探法或通过收集简档数据和 / 或使用所收集的简档数据或通过以其他方式做出一种或多种假设来推导丢失的信息就可以执行某种程度的优化。例如，假设脚本可由运行时引擎接收以作为网页或其它文档的一部分来执行。运行时引擎可使用解释器来执行脚本。在一次或多次脚本的执行期间可由解释器来收集关于该脚本的简档信息。简档信息可被用于确定脚本的哪些部分可通过使用编译器编译这些部分并执行得到的机器代码，而非解释脚本的这些部分，来被更高效地执行。这样，使用解释器可以执行脚本的一些部分，而使用编译器可以编译该脚本的其它部分，并且可以执行所生成的编译的代码。脚本的经编译的部分（例如本机机器代码）可以被存储以供以后在再次执行脚本的这些部分时使用。

[0022] 类似地，可以确定脚本的启动部分。可以对脚本的启动部分加标志，这样，在随后的脚本运行中，在处理非启动部分之前可以由运行时引擎处理该启动部分。

[0023] 根据在此所述的主题的各方面，可以执行包括诸如方法或函数或者方法或函数的任意部分之类的任意代码块的经编译的字节代码。可以基于关于在运行时对程序可用的信息的特性的一种或多种假设来优化编译的字节代码。所述假设可以基于简档信息、试探法或任意其它适合的信息。经优化的编译的字节代码可以包括基于在所述经编译的字节代码中所做的特定优化来预先确定的一个或多个跳出点。对于每个跳出点，可以捕捉一组变量，该组变量保持有解释器在跳出点处恢复执行所需的值。在寄存器分配之后可以创建每个跳出点的跳出表格。每个跳出点的跳出表格可以包括跳出点的每个变量的位置（例如堆栈位

置、寄存器等）。跳出表格还可以包括每个变量的状态（例如在所述变量已经通过类型专用化或其它优化被优化为 int32、双精度等的情况）。

[0024] 在后续经优化的编译的字节代码的执行期间，可以对关于在其上生成所述字节代码的程序将要操作的数据的假设进行测试。响应于确定在其中做出所述优化的一种或多种假设是不正确的，执行可以“跳出”到解释器。就是说，可以停止经编译的代码的执行，并且解释器可以执行与在跳出点处开始的经优化的编译的字节代码所对应的未经优化的字节代码，所述跳出点对应于在经优化的编译的代码中的在其处确定所述一种或多种假设为假的点。当跳出发生时，解释器所需的变量的状态可以通过检查跳出点的跳出表格来被重新物化 (rematerialized)。跳出点的跳出表格可以为变量提供一个值（对于常量）或位置。跳出表格可以提供解释器在该跳出点恢复执行所需的每个变量的状态（对于类型专用化）。在重新物化了解释器继续执行所需的变量的状态之后，解释器可以在跳出位置处恢复控制以承载当前正在执行的函数的执行。

[0025] 根据在此所述的主题的一些方面，可以接收以动态语言编码的源代码以供执行。源代码可以是任意代码块，例如方法或函数或者方法或函数的任意部分。源代码可被解析以生成经解析的源代码。经解析的源代码被转换成字节代码。可以使用解释器来一次或多次解释所述字节代码。可以生成关于字节代码的使用模式的简档信息。可以存储所述简档信息。简档信息可被分析以确定与字节代码接收到的部分相关联的情况。该字节代码部分可被即时 (JIT) 编译成经编译的字节代码部分，作为所确定的情况的结果。可存储经编译的字节代码部分。字节代码部分可稍后在解释器处所接收的字节代码中再次被接收。与所接收到的字节代码部分相对应的经编译的字节代码部分可被定位在存储中。可对位于存储中的经编译的字节代码部分执行至少一个情况检查。如果所述至少一个情况检查通过，则执行经编译的字节代码部分而非解释所接收到的字节代码部分。经编译的字节代码部分可以包括基于假设的优化。所述假设可以基于简档数据、试探法或所提供的其它信息。当执行经优化的代码时，可以执行情况检查。响应于确定所述优化所基于的假设是不正确的，经编译的代码的执行可以结束。代码的执行可以在字节代码中的一个位置处恢复，所述位置类似于在所述假设被确定为假的那点。

[0026] 在系统实现的一个示例中，可以提供运行时引擎。运行时引擎可以包括下述项的全部或一些：解析器、字节代码生成器、执行控制器、解释器、JIT 编译器、简档生成器以及跳出引擎。解析器可以接收用动态语言编码的源代码并可以解析该源代码来生成经解析的源代码。字节码生成器可以将经解析的源代码转换成字节代码。解释器可以被配置为当由执行控制器启用时解释所述字节代码一次或多次。JIT 编译器可被配置成当由执行控制器启用时，编译字节代码。简档生成器可以从字节代码或从源代码中生成简档信息并可以存储所述简档信息。执行控制器可以启用解释器或 JIT 编译器来基于简档信息执行它们各自的功能。

[0027] 随后可以在经编译的代码中做出优化。可以将跳出点插入在经编译的代码中预先确定的点处。可以基于特定优化的特性来预先确定跳出点。对于每个跳出点，可以捕捉一组变量，该组变量保持有使得解释器在程序中的该点处恢复执行的值。可以存储对应于预先确定的跳出点的变量的位置。在后续程序执行中，如果优化所基于的推断证明是不正确的，可以提供后退或跳出机制。可以通过检查与假设相关联的情况来确定推断是不正确的。

根据在此所述的主题的各方面，机器代码执行器可以检查相关联的情况。响应于确定所述推断是不正确的（例如通过机器代码执行器检查情况），根据在此所述的主题的各方面，经编译的代码的执行可以停止，解释器所需的变量可以被重置为在预先确定的跳出点处所保存的值并且可以在解释器处恢复执行。解释器可以在与不正确的假设相关联的跳出点处恢复执行。

[0028] 根据在此所述的主题的各方面，跳出引擎可以通过将与机器代码相关联的变量的状态转换成解释器所需的状态来还原相关联的活动的变量的状态。可以在编译时间（例如由 JIT 编译器）来确定跳出点。在存在依赖于做出的假设的路径的任何地方出现跳出点。每个跳出点可以具有与其相关联的表格，该表格包括关于为了解释器正确执行必须被复活的该组解释器变量的信息，如何执行转换以将变量值从与机器代码相关联的值转换成与解释器相关联的值的描述以及在字节代码中在其处恢复解释的位置。

[0029] 经编译的代码执行中的不再被需要因而非活动的变量可能需要为了解释器而被复活。就是说，一个或多个变量的活跃度可能需要被延长至跳出点以使得该变量所持有的信息对于解释器可用。可以执行优化以最小化对所需的寄存器的数目的影响。例如，如果已知变量值在特定跳出点处为常量值，该常量可以被重新排序以便所述变量在该跳出点处不再需要是可用的。类似地，如果已知变量在特定跳出点处具有与另一变量相同的值，则仅使得一个包含该值的变量在该跳出点处对于解释器可用。

[0030] 将程序执行从经编译的代码变换为经解释的代码

[0031] 图 1 示出根据在此所述的主题的一些方面的 web 浏览器环境 100。如在图 1a 中所示，环境 100 可以包括下述一个或多个：计算设备 102、服务器 104、网络 106 以及浏览器应用 108。Web 浏览环境 100 可以包括本领域中已知的其它组件。

[0032] 计算设备 102 可以是任何类型的固定或移动计算设备，包括桌面计算机（例如，个人计算机等）、移动计算机或计算设备（例如，Palm® 设备、RIM Blackberry® 设备、个人数字助理 (PDA)、膝上型计算机、笔记本计算机、平板计算机（例如，Apple iPad™）、上网本等等）、移动电话（例如，蜂窝电话、智能电话，诸如 Apple iPhone、Google Android™ 手机、Microsoft Windows® 手机等）或其它类型的移动设备。服务器 104 可在包括一个或多个服务器的一个或多个计算机系统中实现，其可以是在此描述的任何类型的计算设备或能够实现在此描述的相应功能的以其它方式知晓的计算设备。

[0033] 计算设备 102 和服务器 104 可通过网络 106 来通信地耦合。网络 106 可包括一个或多个通信链路和 / 或通信网络，诸如 PAN（个域网）、LAN（局域网）、WAN（广域网）、或网络的组合，诸如因特网。可使用各种链路来将计算设备 102 和服务器通信地耦合到网络 106，该各种链路包括有线和 / 或无线链路，如 IEEE 802.11 无线局域网 (WLAN) 无线链路、全球微波互联接入 (Wi-MAX) 链路、蜂窝网络链路、无线个人区域网络 (PAN) 链路（例如，Bluetooth™ 链路）、以太网链路、USB 链路等等。

[0034] 浏览器应用 108 可以是可在计算设备 102 上执行的程序。浏览器应用 108 可使得网络信息资源能被检索、呈现和遍历。可使用网络地址（诸如统一资源标识符 (URI)）来通过浏览器应用 108 检索信息资源或对象。信息资源的示例包括网页、图像、视频，以及其它形式的内容。出现在信息资源中的超链接使得用户能够容易地将他们的浏览器导航到相关的资源。浏览器应用 108 的示例包括由华盛顿州雷蒙德市的微软

公司开发的 Internet Explorer®、由加利福尼亚州芒廷维市的 Mozilla 公司开发的 Mozilla Firefox®、由加利福尼亚州库珀蒂诺市的 Apple 公司开发的 Safari®，以及由加利福尼亚州芒廷维尤市的 Chrome 公司开发的Google®。

[0035] 浏览器应用 108 可以通过网络 106 从服务器 104 检索文档 112。文档 112 可以是包括标记语言的代码的 web 文档，标记语言诸如超文本标记语言 (HTML)、动态 HTML(DHTML)、可扩展 HTML(XHTML)、可扩展标记语言 (XML) 等。文档 112 可以包括 DOM(文档对象模型) 对象 114 和一个或多个脚本 116。DOM 对象 114 可以包括根据 DOM 规范在文档 112 中所表示的一个或多个对象，该 DOM 规范是用于表示对象并与对象进行交互的跨平台且独立于语言的规范。DOM 对象 114 可包括被直接包括在文档 112 中的对象、和 / 或由文档 112 引用的并分开地从服务器 104 或其它服务器检索的对象。脚本 116 包括根据动态语言 (例如，JavaScript、VBScript、AJAX、Python、Perl 等) 来格式化的代码，该格式化的代码使得能够对 DOM 对象 114 做出改变，包括基于诸如用户输入、环境情况 (例如，一天中的时间或其它变量) 等因素的改变。脚本 116 的代码可在进行中访问和修改 DOM 对象 114 的对象，而无需返回到服务器 104。

[0036] 浏览器应用 108 可以接收 (例如加载) 文档 112。浏览器应用 108 可包括浏览器引擎 (例如，布局引擎或呈现引擎)，该浏览器引擎格式化文档 112 的信息并显示经格式化的信息。例如，如图 1a 显示的，浏览器应用 108 可基于由计算设备 102 的显示器 110 显示的文档 112 来生成页面 118。浏览器应用 108 可被配置成执行一个或多个脚本 116，该一个或多个脚本 116 被嵌入在文档 112 中或与文档 112 分开但与文档 112 相关联。

[0037] 图 1b 示出根据在此所述的主题的一些方面的系统 200 的框图。系统 200 可包括一个或多个计算机或计算设备，例如包括诸如处理器 143 等之类的一个或多个处理器、存储器 145、执行控制器 309、解释器 311、诸如 JIT(即时) 编译器 313、存储 315、机器代码执行器 317 和跳出引擎 337 之类的编译器的计算设备 102。执行控制器 309、解释器 311、诸如 JIT(即时) 编译器 313 的编译器、存储 315、机器代码执行器 317 和跳出引擎 337 可被实现为一个或多个程序模块，当所述程序模块被加载到存储器 145 中时使得所述一个或多个处理器 143 等分别执行归因于执行控制器 309、解释器 311、诸如 JIT(即时) 编译器 313、存储 315、机器代码执行器 317 和跳出引擎 337 的动作。

[0038] 执行控制器 309、解释器 311 和 / 或 JIT 编译器 313 可以接收从源代码生成的字节代码。源代码可以是任意以动态语言编写的源代码，例如但不局限于动态脚本语言 (诸如但不限于 JavaScript、VBScript、Python 等等。可以解析源代码并从经解析的源代码中生成字节代码。基于字节码 325 和简档、试探法或其它信息，执行控制器 309 可启用解释器 311 和 JIT 编译器 313 之一来对字节代码 325 进行操作。解释器 311 可被配置成，当由从执行控制器 309 接收到的解释器控制信号启用时解释字节代码 325。JIT 编译器 313 可被配置成，当由从执行控制器 309 接收到的编译器控制信号启用时编译字节代码 325。

[0039] 当由解释器控制信号启用解释器 311 时，解释器 311 可以解释并执行未经优化的字节代码 325。解释器 311 可以被实现为 JavaScript 解释器、VBScript 解释器、Python 解释器或用于在此其它地方提到的或以其它方式所知的另一动态语言或动态脚本语言的解释器。通过该方式，源代码可至少部分地由解释器 311 的操作来执行。类似地，响应于接收

到启用编译器控制信号，JIT 编译器 313 可以编译字节代码 325。JIT 编译器 313 可被实现为 JavaScript 编译器、VBScript 编译器、Python 编译器或用于在此其它地方提到的或以其它方式所知的另一动态语言或动态脚本语言的编译器。JIT 编译器 313 被称为“即时”编译器，这是因为由于需要经编译的字节代码（即，要被紧急执行的），特定的字节代码部分可由 JIT 编译器 313 编译，而非在执行之前预先编译完整的字节代码 325。JIT 编译器 313 可以生成具有机器可执行代码或指令形式的经编译的字节代码 333。

[0040] JIT 编译器 313 可以基于如上所述的假设执行代码优化。JIT 编译器 313 可以在它生成的经编译的字节代码中插入一个或多个预先确定的跳出点。对于每个跳出点，诸如跳出表格 303 等之类的跳出表格可以被创建，在其中，可以记录变量的位置、在字节代码 325 中与跳出点的位置相对应的位置以及其它信息。跳出表格 303 等可以描述在堆栈上或寄存器中哪里可以找到变量。JIT 编译器 313 可以生成一个或多个跳出表格并可以将它们与经优化的编译的字节代码（例如作为机器代码）一起保存。可以通过编码跳出表格中的信息来避免对变量的生存期的非必要的延长来优化跳出表格。例如，如果已知变量 x 的值在跳出时间是常量 10，x ＝ 10 可以被编码在跳出表格中，这样，值 10 就不需要在存储器或寄存器中被复活。类似地，当超过一个变量在跳出点处具有相同的值（例如 x ＝ y），如果该信息被编码在跳出表格中（例如跳出表格中的相同位置可以被用于 x 和 y），则对于具有相同值的所有变量来说，该相同的位置可被用在跳出表格中。

[0041] 这些技术可以使得寄存器分配更加有效。经优化的编译的字节代码可以被作为用于在由系统 200 后续执行程序期间访问的经编译的字节代码 333 来存储在存储 315 中。

[0042] 当执行经优化的编译的字节代码时，可由机器代码执行器 317（它可以是诸如处理器 143 等的一个或多个处理器）接收经优化的编译的字节代码 333。执行经编译的字节代码可以与堆栈 335 上的编译器框架 341 相关联。在每个跳出点处可以确定所述优化所基于的底层的一种或多种假设是有效还是无效的。响应于确定所述假设是有效的，经优化的编译的代码可以继续执行。响应于确定所述假设是无效的，经优化的编译的代码的执行可以被停止。可以将该跳出点的跳出表格传递给跳出引擎 337。跳出引擎 337 可以还原解释器所需的变量的状态（例如值）。跳出引擎可以为堆栈 335 创建新的框架（例如解释器框架 339）。如果解释器所需的变量已变成非活动的，则跳出引擎 337 可以通过将变量的状态从非活动改变到活动来复活该变量。跳出引擎 337 可以实例化解释器 311 的一个实例，向解释器传递在未经优化的字节代码中与在经优化的编译的代码中的跳出点相对应的位置以及包括所有被还原的活动变量的值的新创建的解释器框架。因此这样，从其生成字节代码 325 的源代码可以由 JIT 编译器 313 和机器代码执行器 317 的操作来部分执行并由解释器 311 来部分执行。

[0043] 图 1c 示出根据在此所述的主题的各方面的系统 300 的另一示例。系统 300 可以包括诸如处理器 142 等的一个或多个处理器、存储器 144 以及包括下述一个或多个项的运行时引擎：引擎接口 302、解析器 304、字节代码生成器 306、执行控制器 308、解释器 310、诸如但不限于 JIT（即时）编译器 312 之类的编译器、存储 314、机器代码执行器 316、跳出引擎 336 以及脚本库 318。

[0044] 如在图 1c 中所示的，引擎接口 302 可接收脚本源代码 208。引擎接口 302 可以是在场的或不在场的。解析器 304 可以被配置作为到运行时引擎的接口，而不是使得引擎接

口 302 在场。当在场时，引擎接口 302 是提供用于将主机与图 1c 的运行时引擎对接的一个或多个方法的通信接口。根据在此所述的主题的一些方面，可以根据由华盛顿州雷蒙德市的微软公司所开发的 IActiveScript 来实现引擎接口 302。引擎接口 302 可以为解析器 304 提供源代码 208。

[0045] 解析器 304 可以接收并解析源代码 208。解析器 304 可对源代码 208 执行令牌生成或词法分析，使得源代码 208 被格式化成符号或令牌。解析器 304 可对令牌执行差错校验以确定是否形成可允许的表达式、句法差错不存在等。解析器 304 可以将经解析的源代码作为经解析的源代码（未示出）输出。经解析的源代码可具有任何合适的形式，包括由解析器 304 生成为 AST（抽象句法树）代码，如本领域的技术人员知晓的，该 AST 代码包括源代码 208 的抽象句法结构的树表示。

[0046] 字节代码生成器 306 可以接收经解析的源代码。字节代码生成器 306 可被配置成将经解析的源代码转换成字节代码，其包括被配置成用于解释器的高效执行以及用于进一步编译成机器代码的指令集。所生成的字节代码可以将经解析的源代码表示为数字代码和相应的可选参数。字节代码生成器 306 可以输出经生成的字节代码（未示出）。字节代码可具有任何合适的形式，包括由字节代码生成器 306 生成的如本领域的技术人员知晓的 p- 代码（便携式代码）形式。

[0047] 执行控制器 308、解释器 310 和 JIT 编译器 312 每个都可以接收所生成的字节代码。解释器 310 可以包括简档生成器 204。简档生成器 204 可被配置成分析所生成的字节代码来收集有关源代码 208 的统计数据以及进一步的信息。简档生成器 204 可生成简档信息 320，其可包括所收集的信息并可被存储在存储 314 中。

[0048] 执行控制器 308 可以访问简档信息 320，并且可以被通信地耦合到解释器 310 和 JIT 编译器 312。基于所生成的字节代码和简档信息 320，执行控制器 308 可启用解释器 310 和 JIT 编译器 312 之一来对所生成的字节代码进行操作。解释器 310 可被配置成，当由从执行控制器 308 接收到的解释器控制信号启用时解释所生成的字节代码。JIT 编译器 312 可被配置成，当由从执行控制器 308 接收到的编译器控制信号启用时编译所生成的字节代码。例如，在源代码 208 的第一次执行期间，简档信息 320 可能还不存在。在这种情况下，执行控制器 308 可启用解释器 310 来解释所生成的字节代码并生成简档信息 320。在源代码 208 随后的执行期间（例如，在源代码 208 的相同的第一次执行期间的稍后，和 / 或在源代码 208 的接下来的执行期间），根据简档信息 320，执行控制器 308 可启用解释器 310 来解释源代码 208 的各部分，并可启用 JIT 编译器 312 来编译源代码 208 的其它各部分。

[0049] 当由解释器控制信号启用解释器 310 时，解释器 310 可以解释并执行所生成的字节代码。解释器 310 可被实现为 JavaScript 解释器、VBScript 解释器、Python 解释器或用于在此其它地方提到的或以其它方式所知的另一动态语言的解释器。通过该方式，源代码 208 可至少部分地由解释器 310 的操作来执行。

[0050] 当 JIT 编译器 312 由编译器控制信号启用时，JIT 编译器 312 可编译所生成的字节代码。JIT 编译器 312 可被实现为 JavaScript 编译器、VBScript 编译器、Python 编译器或用于在此其它地方提到的或以其它方式所知的另一动态语言的编译器。JIT 编译器 312 被称为“即时”编译器，这是因为由于需要经编译的字节代码（即，要被紧急执行的），特定的字节代码部分可由 JIT 编译器 312 编译，而非在执行之前预先编译完整的字节代码。JIT

编译器 312 可以生成具有机器可执行代码或指令形式的经编译的字节代码 332。

[0051] JIT 编译器 312 可以基于如上所述的假设执行代码优化。JIT 编译器 312 可以在它生成的经编译的字节代码中插入一个或多个预先确定的跳出点。对于每个跳出点，诸如跳出表格 303 等之类的跳出表格可以被创建，在其中，可以记录变量的状态、在所生成的字节代码中与跳出点的位置相对应的位置。跳出表格可以描述在堆栈上或机器代码的寄存器中哪里可以找到变量。JIT 编译器 312 可以生成表格并将它们与经优化的编译的字节代码 332（例如机器代码）一起保存。经优化的编译的字节代码 332 可以被作为用于在由系统 300 后续执行程序期间访问的经编译的字节代码 332 来存储在存储 314 中。

[0052] 当执行经优化的编译的字节代码时，可由机器代码执行器 316（它可以是诸如处理器 142 等的一个或多个处理器）接收经优化的编译的字节代码 332。在每个跳出点处可以检查所述优化所基于的底层的一种或多种假设是有效还是无效的。响应于确定所述假设是有效的，经优化的编译的字节代码 332 可以继续执行。响应于确定所述假设是无效的，经优化的编译的代码的执行可以被停止。可以将该跳出点的跳出表格传递给跳出引擎 336。跳出引擎 336 可以还原解释器所需的变量的状态（例如值）。如果解释器所需的变量已变成非活动的，跳出引擎 336 可以通过将变量的状态从非活动改变到活动来复活该变量。跳出引擎 336 可以实例化解释器 310 的实例，向解释器传递在未经优化的字节代码中与在经优化的编译的代码中的跳出点相对应的位置、变量和它们的位置。因此这样，从其生成字节代码的源代码可以由 JIT 编译器 312 和机器代码执行器 316 的操作来部分执行并由解释器 310 来部分执行。

[0053] 跳出的使用可以允许多次优化。在许多动态编程语言中的变量可以是日期、字符串、数组、整数等等。在一些动态语言中，为变量创建对象，并且存储在寄存器中并随后存储在表格中的位置包括指向该对象的指针。对象自身可以存储对象的描述和对象的数据。这样，如果变量 a 是字符串，可以为该字符串创建对象，指向该对象的指针可以被存储，且对象自身可以标识变量 a 为字符串并可以指定该字符串自身（例如“你好世界”（“Hello World”））。一种用于在当基于简档数据假设变量为整数的情况下的方案的优化是将整数自身存储在变量中以取代指向为该整数创建的对象的指针或表示并描述该整数对象的经编码的值。一种直接对整数对象进行编码的方案是使用单词的位之一来存储标记该值为整数的指示符。这样，在 32 位处理器的情况下，32 位的一个位可以指示其它 31 位的内容表示整数值或可指示其它 31 位的内容不表示整数值。

[0054] 图 1d 示出以 JavaScript 编写的源代码 152 的示例 150，在其中，函数 foo 执行两个变量 a 和 b 的加法并返回加法操作的结果。在示例 150 中显示的字节代码 154 是从该示例的源代码 152 中生成的字节代码。在行 #1，将第一自变量 (a) 加载到解释器临时变量 t1。在行 #2，将第二自变量 (b) 加载到解释器临时变量 t2。在行 #3，临时变量 t1 的内容和临时变量 t2 的内容被相加并放置到解释器临时变量 t0 中。在行 #4，返回临时变量 t0 的内容。临时变量 t1、临时变量 t2 和临时变量 t0 的内容在 JavaScript 中可以无论怎样都是合法的，包括浮点数、整数、字符串等等。

[0055] 在图 1d 中，所生成的代码 156 示出表示编译字节代码 154 并基于简档数据对其进行整数加法优化的结果的伪代码。当字节代码 154 被编译时，并不知道变量 a 和 b 的类型是什么。变量 a 和 / 或变量 b 可以是字符串、日期、整数等等。如果变量 a 和 b 是日期，将

执行一种加法操作，如果变量 a 和变量 b 是字符串，串联操作将准备就绪，并且如果变量 a 和变量 b 是浮点数，则将调用浮点加法操作。如果期望变量 a 和变量 b 很可能是整数（例如通过运行解释器多次并发现变量 a 和变量 b 总是整数）是合理的，那么编译的代码的乐观优化可以执行整数加法，如由所生成的代码 156 所示，该乐观优化针对整数加法进行了优化。由于在当变量 a 和变量 b 是日期、字符串或非整数数字的情况下通过不提供代码路径减少了执行的代码量，所述代码被优化了。

[0056] 在所生成的代码 156 的行 #1，将第一自变量 (a) 加载到寄存器 reg1。在这点（跳出点 #1）处，检查 reg1 的内容以查看 reg1 的内容是否真的是整数。响应于 reg1 的内容不是整数，（行 #2），所生成的代码的执行被停止，并且调用跳出引擎（在所生成的代码 156 的行 #3 处执行跳出指令）。在解释器中在字节代码 154 的行 #2 处继续程序执行，因为字节代码 154 的行 #1 已经被执行。t1 的内容已经被放置在 reg1 中。t2 和 t0 还不是活动的，并且不必被复活。发送给跳出引擎的表格的内容被示出在图 1d 的表格 1 的内容 158 中。表格 1 的内容 158 描述了跳出点 1 的执行在字节代码 154 的行 #2 处恢复，变量 t1 的值被存储在 reg1 中，而变量 t2 和 t0 当前并不是活的。这意味着变量 t2 和 t0 是不活动的且不需要被复活。

[0057] 如果相反，reg1 的内容是整数，则所生成的代码 156 在所生成的代码 156 的行 #4 处继续执行。在行 #4 处，自变量 b 被加载到寄存器 reg2 中。在所生成的代码 156 的行 #5 处，检查 reg2 的内容以确定它是整数还是非整数。如果 reg2 的内容是非整数，在行 #6 执行跳出指令。所生成的代码 156 的执行停止，且在解释器中在字节代码 154 的行 #3 处执行恢复。变量 a 和 b 是活动的。存储在 t1 中的变量 a 的值驻留在 reg1，而存储在 t2 中的变量 b 的值驻留在 reg2。t0 不是活的。该信息被放置在表格 2（表格 2160 的内容）中。

[0058] 如果相反，reg2 的内容是整数，则所生成的代码 156 在所生成的代码 156 的行 #7 处继续执行。将 reg1 和 reg2 的内容相加并放置在寄存器 reg3 中。在行 #7，reg1 和 reg2 相加的结果被放置在寄存器 reg3 中。在行 #8，如果 reg3 溢出，则到达第三跳出点。在第三跳出点，在行 #9，跳出被定向到字节代码 154 的行 #3 以通过不丢失信息的方式来重做操作。相关联的表格 Table3162 的内容与表格 2 是相同的。为了避免丢失信息，整数值可以被转换成浮点数，并且在解释器中可以执行浮点加法。如果相反，reg3 不溢出，所生成的代码 156 在所生成的代码 156 的行 #10 处继续执行并返回 reg3 中的值。

[0059] 在上述示例中，JavaScript 指针的一个位可以被设定为指示符值，该指示符值指示了剩余的 31 位表示整数值。为了在整数上执行数学运算，可以从存储了指针的 32 位寄存器中提取表示整数的 31 位并可以执行所述运算。随后可以在经解码的整数值上执行进一步的数学运算。一旦数学运算的值或结果需要被存储或传递到当前编译范围之外的进程，可以将其重新编码成 JavaScript 对象指针，将所述指示符位设定回指示符值。例如，根据在此所述的主题的一些实施例，在所生成的代码 156 的行 #10 处返回 reg3 之前，可以撤消整数类型的专用化并重新编码该值。（为了清楚起见，编码 / 解码步骤并没有包括在伪代码示例中。）表格还可以捕捉 t1 是在 reg1 中的信息。因为该信息是被解码的，它可以被重新编码成 JavaScript 值，将指示符位重置为指示其它 31 位不是整数值。类似地，64 位浮点值可以被表示为对象，这样，对于在 reg1 中的浮点数来说，可以做出 reg1 的内容是浮点数的检查。如果是，来自 JavaScript 数字的浮点数据被加载到浮点寄存器中并且最后可以创建

新的 JavaScript 数字，在该新的 JavaScript 数字中，在要返回给调用者的寄存器中放置结果数字、放置指向 JavaScript 数字的指针。

[0060] 图 1e 示出为图 1d 的相同的 JavaScript 源代码（源代码 152）的浮点数的加法优化的经编译的代码的示例 170。源代码和字节代码 154 两者是相同的，例如 170 与例如 150 相同，然而，仅仅为了示例 170，假设期望变量是浮点数而不是整数是合理的。将理解的是，第一跳出点和第一跳出点的对应的表格信息与针对整数加法优化的所生成代码 156 相同。

[0061] 在示例 170 的行 #1 处，变量可以被加载到寄存器 reg1 中。在行 #2 处检查 reg1 的内容以确定 reg1 的内容是否指向 JavaScript 数字。响应于确定 reg1 的内容不是指向 JavaScript 数字，可在行 #3 调用跳出引擎。存储在该跳出点（跳出点 #1）的表格中的跳出信息与先前示例 150 中的相同。如果 reg1 的内容指向 JavaScript 数字，在行 #4 处，可以通过使用 reg1 访问来自 JavaScript 数字的浮点数据来从 JavaScript 数字中查明浮点值，并将浮点数据加载到浮点寄存器 floatReg1 中。因为不再需要 reg1，编译器可以在行 #5 处重用 reg1 来加载变量 b。在行 #6 处，可以检查 reg1 以确定其内容是否指向 JavaScript 数字。如果它没有，在行 #7 处可以调用跳出引擎。在跳出点 #2，为了复活 t1，可以从 floatReg1 中查明变量 a 的值。

[0062] 跳出引擎可以确定变量 a 是浮点数字，并可创建 JavaScript 数字对象，并可放置 t1 中的 JavaScript 数字的位置以允许解释器开始从字节代码 154 的行 #3 处执行，因为解释器处理不是浮点值的 JavaScript 对象。在该点处，t2 驻留在 reg1 中，而 t0 不是活的。如果相反，在行 #6 处，reg1 的内容确实指向 JavaScript 数字，可以通过使用 reg1 访问来自 JavaScript 数字的浮点数据来从 JavaScript 数字中查明浮点值，并在行 #8 将浮点数据加载到浮点寄存器 floatReg2 中。在行 #9，可以执行两个浮点数字的加法并可将结果存储在 floatReg3 中。在此不需要跳出，因为 JavaScript 将数字运算定义为正被双精度浮点运算执行，因此，任何浮点溢出都将表现出按每 JavaScript 惯例的行为。因为不再需要 reg1，在行 #10 处编译器可以重用 reg1 以保持指向所构造的 JavaScript 数字对象的位置的指针，所述 JavaScript 数字对象持有将两个浮点值相加的结果。在行 #11，可将 reg1 返回给调用者。

[0063] 当第一函数（例如函数 foo）调用第二函数（例如函数 bar）时，可以应用另一种被称为内联 (in-lining) 的优化。在这种情况下，函数 bar 的代码可以以顺序次序在函数 foo 的代码内部被写出。如果在内联的函数内部出现跳出，则调用函数和被调用函数两者的状态可都被还原，并且这两个函数可以运行。可以创建被调用函数的表格以及调用函数的表格。

[0064] 当以诸如 JavaScript 之类的动态语言做出函数调用时，编译器不能确定在编译时哪个函数正被调用。虽然脚本可以指示函数 foo 正被调用，在运行时，被称为 foo 的函数可以改变。这样，就不能做出关于在编译时哪个函数正被调用的假设。内联在调用函数的内部直接生成被调用函数的代码。当该代码被执行时，可以动态地做出经内联的函数是应该实际被调用的函数的检查。根据在此所述的主题的各方面，如果正被调用的函数不是经内联的函数，则可做出对跳出引擎的调用。在图 1f 中，示例 171，JavaScript 函数 bar172 以及 JavaScript 函数 foo173 被定义。经编译的代码在经编译的代码 174 中示出。

[0065] 编译器可以做出下述优化。如果 bar 被调用，则声明 var c = a+b ;在函数 foo173

中是非必要的，因为 bar 总是返回 10。函数 foo 返回 0，因为 bar 返回值 10。但由于 bar 可以改变，可做出一个检查来查看 bar 是否已经改变。如果 bar 已经改变，可以做出对跳出引擎的调用。如果做出对跳出引擎的调用，解释器将需要变量 c。为了使得 c 可用，将需要 a+b。a+b 被存储在 reg3 中，因此，不能释放 reg3 以供重用直到跳出发生之后。这样，以其它方式将变得非活动的变量可以被维持在活动状态。

[0066] 在图 1g 中的示例 180 中，JavaScript 函数 bar182 以及 JavaScript 函数 foo183 被定义。函数 bar182 具有两个自变量：变量 a 和函数 func。脚本 184 调用具有 a ＝ 2 和 func ＝ foo 的函数 bar182。从脚本 184 生成的字节代码在字节代码 185 中被示出。

[0067] 在行 #1 处，函数 bar185 的字节代码将变量 a 的值 (2) 放置在 t1 中。在行 #2 处，常量 100 被放置在 t2 中，在行 #3 处，函数 foo 被放置在 t3 中，而在行 #4 处，用自变量 2 和 100 来调用函数 foo。在行 #5 处，将用自变量 2 和 100 调用 foo 的结果与 2 相加并将结果放置在被返回的 t0 中。函数 foo 的经编译的字节代码在图 1d 中被示为所生成的代码 156。将函数 foo183 内联到函数 bar182 的经编译的代码在图 1g 中被示为所生成的代码 186。在第一跳出点前，BarNativeCode(Bar 本机代码）是堆栈的顶层框架。当在第一跳出点处调用跳出引擎时，调用跳出引擎的新框架将被放置在堆栈的顶部。当跳出引擎创建 foo 的存储的经解释的框架并开始在解释器上运行它时，顶部的 4 个框架将是 foo 经还原的被解释的框架，下一个将是跳出引擎的框架，下一个是具有内联的 foo 和调用者框架的 Bar NativeCode（本机代码）的框架。在经还原的 foo 完成正被解释之后，跳出引擎可以创建 bar 的经还原的被解释的框架并可以开始在解释器上运行它。从上至下堆栈将出现：Bar 经还原的被解释的框架、跳出引擎的框架、具有内联的 foo 和调用者框架的 Bar NativeCode 的框架。最后，在经还原的 bar 完成解释之后，跳出引擎可以返回到 bar 本机代码，并且 bar 本机代码可以返回到调用者。在这个点，堆栈的顶层框架将是调用者框架。

[0068] 内联代码 188 和经内联的代码 189 表示在所生成的代码 186 中来自已经被内联到函数 bar 中的函数 foo 的代码。在函数 foo 已经被内联到函数 bar 中之后，函数 foo 可以改变。到 bar 函数字节代码 185 的第一跳出检查 foo 是正确的函数。该函数被加载到 reg3 中并检查 reg3 以查看函数是否是 foo。如果是，经编译的代码的执行继续。如果不是，执行在解释器中恢复。在第一跳出点处调用跳出引擎，并且执行在函数 bar185 的行 #4 处恢复。这是一个简单的跳出，因为执行是在内联外部发生的，这样在还原 t1、t2 和 t3 之后，执行可以在解释器中继续。如果该函数是 foo，执行继续并且执行经内联的代码 188。在经内联的代码 188 中，如果 reg1 的内容不是整数，则在第二跳出点处调用跳出引擎，并且执行在 foo（字节代码 154）中的行 #2 处恢复。由于 bar 实际上正在被执行，到函数 bar 的行 #5 的跳出可以被很好完成。在 foo 完成执行之后，foo 的结果可以被放置在 t4 中。可以做出到 bar 的行 #5 的跳出，t1 在 reg1 中，并且 t4 将包含从跳出到 foo 的解释器返回值。该信息可以被编码到 2 个链接在一起的表格中，以便这两个跳出将一个接一个发生。在加法溢出时，可以在内联代码 189 中执行类似的处理。将可以理解，如上所述，跳出在一行中出现一次，而框架被每次复活一个框架。当返回给调用者时，调用者的框架被复活。

[0069] 在第一跳出点前，BarNativeCode 是堆栈的顶层框架。当在第一跳出点处调用跳出引擎时，调用跳出引擎的新框架将被放置在堆栈的顶部。当跳出引擎创建 foo 的存储的经解释的框架并开始在解释器上运行它时，顶部的 4 个框架将是 foo 经还原的被解释的框

架，下一个将是跳出引擎的框架，下一个是具有内联的 foo 的 Bar NativeCode 的框架。在经还原的 foo 完成正被解释之后，跳出引擎可以创建 bar 的经还原的被解释的框架并可以开始在解释器上运行它。从上至下堆栈将出现 :Bar 经还原的被解释的框架、跳出引擎的框架、具有内联的 foo 和调用者框架的 Bar NativeCode 的框架。最后，在经还原 bar 完成解释之后，跳出引擎可以返回到 bar 本机代码，并且 bar 本机代码可以返回到调用者。在这个点，堆栈的顶层框架将是调用者框架。

[0070] 图 2 示出了可以将以动态语言编写的程序的执行从经编译的代码变换成经解释的代码的方法 250。方法 250 可以由诸如如上所述的系统 100、系统 200 和 / 或系统 300 之类的系统来实现。所描述的顺序是可选的。并不是所有的动作都要被采用。在 252，可以由计算设备的处理器接收字节代码。在 254，可以做出关于程序可以在其上操作的数据的一种或多种假设。在 256，基于所述假设，可以基于所述一种或多种假设为所接收的字节代码的一部分生成经优化的编译的代码。在预置的跳出点处可以捕捉变量值位置。还可以捕捉在字节代码中的对应位置。在 260，可以执行经优化的编译的代码。变量值可以被保存在变量值位置中。在 262，可以检查经编译的代码中的优化所基于的假设的有效性。在 264，响应于确定所述假设还是有效的，经编译的代码的执行可以继续。在 266，响应于确定所述假设是无效的，经编译的代码的执行可以在跳出点处停止。在 268，来自所捕捉位置的变量值可以被用于还原解释器变量值。在 270，对在字节代码中的所指示的模拟位置处的字节代码的解释可以开始。

[0071] 合适的计算环境的示例

[0072] 为了提供有关本文所公开主题的各方面的上下文，图 3 以及以下讨论旨在提供其中可以实现本文所公开主题的各实施例的合适计算环境 510 的简要概括描述。尽管本文所公开的主题是在诸如程序模块等由一个或多个计算机或其他计算设备执行的计算机可执行指令的通用上下文中描述的，但本领域技术人员将认识到，本文所公开的主题的各部分还能够结合其他程序模块和 / 或硬件和软件的组合来实现。通常，程序模块包括执行特定任务或实现特定数据类型的例程、程序、对象、物理人为产物、数据结构等。通常，程序模块的功能可在各个实施例中按需进行组合或分布。计算环境 510 只是合适的操作环境的一个示例，并且不旨在对此处所公开的主题的使用范围或功能提出任何限制。

[0073] 参考图 3，描述了计算机 512 形式的计算设备。计算机 512 可包括至少一个处理单元 514、系统存储器 516 和系统总线 518。至少一个处理单元 514 可执行被存储在诸如但不限于系统存储器 516 之类的存储器中的指令。处理单元 514 可以是各种可用处理器中的任何一种。例如，处理单元 514 可以是 GPU。这些指令可以是用于实现被描述为由上述一个或多个组件或模块所执行的功能的指令或用于实现上述方法中的一个或多个的指令。也可以使用双微处理器及其他多处理器体系结构作为处理单元 514。计算机 512 可被用于支持在显示屏上呈现图形的系统中。在另一示例中，计算设备的至少一部分可以用在包括图形处理单元的系统中。系统存储器 516 可包括易失性存储器 520 和非易失性存储器 522。非易失性存储器 522 可包括只读存储器 (ROM)、可编程 ROM(PROM)、电可编程 ROM(EPROM) 或闪存。易失性存储器 520 可包括可充当外高速缓冲存储器的随机存取存储器 (RAM)。系统总线 518 将包括系统存储器 516 的系统物理人为产物耦合到处理单元 514。系统总线 518 可以是几种类型的总线结构中的任何一种，包括存储器总线、存储控制器、外围总线、外总线

或局部总线，并且可以使用各种可用总线体系结构中的任一种。计算机 512 可包括处理单元 514 可通过系统总线 518 访问的数据存储。数据存储可包括用于图形呈现的可执行指令、3D 模型、素材、材质等。

[0074] 计算机 512 通常包括各种计算机可读介质，诸如易失性和非易失性介质、可移动和不可移动介质。计算机存储介质可以通过用于存储诸如计算机可读指令、数据结构、程序模块或其它数据等信息的任何方法或技术来实现。计算机存储介质包括但不限于，RAM、ROM、EEPROM、闪存或其它存储器技术、CDROM、数字多功能盘 (DVD) 或其它光盘存储、磁盒、磁带、磁盘存储或其它磁存储设备、或可以用来储存所期望的信息并可由计算机 512 访问的任何其他瞬态或非瞬态介质。

[0075] 将理解，图 3 描述了可充当用户与计算机资源之间的媒介的软件。该软件可以包括可存储在盘存储 524 上的操作系统 528，该操作系统可分配计算机 512 的资源。盘存储 524 可以是通过诸如接口 526 等不可移动存储器接口连接到系统总线 518 的硬盘驱动器。系统应用程序 530 利用由操作系统 528 通过存储在系统存储器 516 或者存储在盘存储 524 上的程序模块 532 和程序数据 534 对资源的管理。可以理解，计算机可用各种操作系统或操作系统的组合来实现。

[0076] 用户可通过输入设备 536 向计算机 512 输入命令或信息。输入设备 536 包括但不限于定点设备，诸如鼠标、跟踪球、指示笔、触摸垫、键盘、话筒等。这些及其他输入设备通过系统总线 518 经由接口端口 538 连接到处理单元 514。接口端口 538 可表示串行端口、并行端口、通用串行总线 (USB) 等。输出设备 540 可与输入设备使用相同类型的端口。提供输出适配器 542 以举例说明存在像监视器、扬声器、以及打印机的需要特定适配器的一些输出设备 540。输出适配器 542 包括但不限于，在输出设备 540 和系统总线 518 之间提供连接的视频卡和声卡。其他设备和 / 或系统和 / 或设备，诸如远程计算机 544，可提供输入和输出两种能力。

[0077] 计算机 512 可以使用到诸如远程计算机 544 之类的一个或多个远程计算机的逻辑连接来在联网环境中操作。远程计算机 544 可以是个人计算机、服务器、路由器、网络 PC、对等设备或其他常见的网络节点，并且通常包括许多或所有以上相对于计算机 512 所描述的元件，但在图 3 中仅示出了存储器存储设备 546。远程计算机 544 可经由通信连接 550 逻辑地连接。网络接口 548 涵盖诸如局域网 (LAN) 和广域网 (WAN) 这样的通信网络，但也可包括其他网络。通信连接 550 是指用来将网络接口 548 连接到总线 518 的硬件 / 软件。通信连接 550 可以在计算机 512 内或外并且包括诸如调制解调器（电话、电缆、DSL 和无线）和 ISDN 适配器、以太网卡等内和外技术。

[0078] 可以理解，所示网络连接仅是示例，并且可以使用在计算机之间建立通信链路的其他手段。本领域的普通技术人员可以理解，计算机 512 或其他客户机设备可作为计算机网络的一部分来部署。在这一点上，本文所公开的主题涉及具有任意数量的存储器或存储单元以及在任意数量的存储单元或卷上发生的任意数量的应用和进程的任何计算机系统。本文所公开的主题的各方面可应用于具有部署在网络环境中的具有远程或本地存储的服务器计算机和客户计算机的环境。本文所公开的主题的各方面也可应用于具有编程语言功能、解释和执行能力的独立计算设备。

[0079] 本文所述的各种技术可结合硬件或软件，或在适当时以其组合来实现。由此，本文

所公开的方法和装置或其特定方面或部分可采取包含在诸如软盘、CD-ROM、硬盘驱动器或任何其他机器可读存储介质等有形介质中的程序代码（即，指令）的形式，其中当程序代码被加载到诸如计算机等机器内并由其执行时，该机器成为用于实现本文所公开的主题的各方面的装置。如此出所用的，术语“机器可读介质”应被用来排除提供（即存储和/或传输）任何形式的传播信号的任何机制。在程序代码在可编程计算机上执行的情况下，计算设备通常将包括处理器、该处理器可读的存储介质（包括易失性和非易失性的存储器和/或存储元件）、至少一个输入设备、以及至少一个输出设备。可例如通过使用数据处理 API 等来利用域专用编程模型各方面的创建和/或实现的一个或多个程序可用高级过程语言或面向对象的编程语言来实现以与计算机系统通信。然而，如果需要，该程序可以用汇编语言或机器语言来实现。在任何情形中，语言可以是编译语言或解释语言，且与硬件实现相结合。

[0080] 尽管用结构特征和/或方法动作专用的语言描述了本主题，但可以理解，所附权利要求书中定义的主题不必限于上述具体特征或动作。更确切而言，上述具体特征和动作是作为实现权利要求的示例形式公开的。

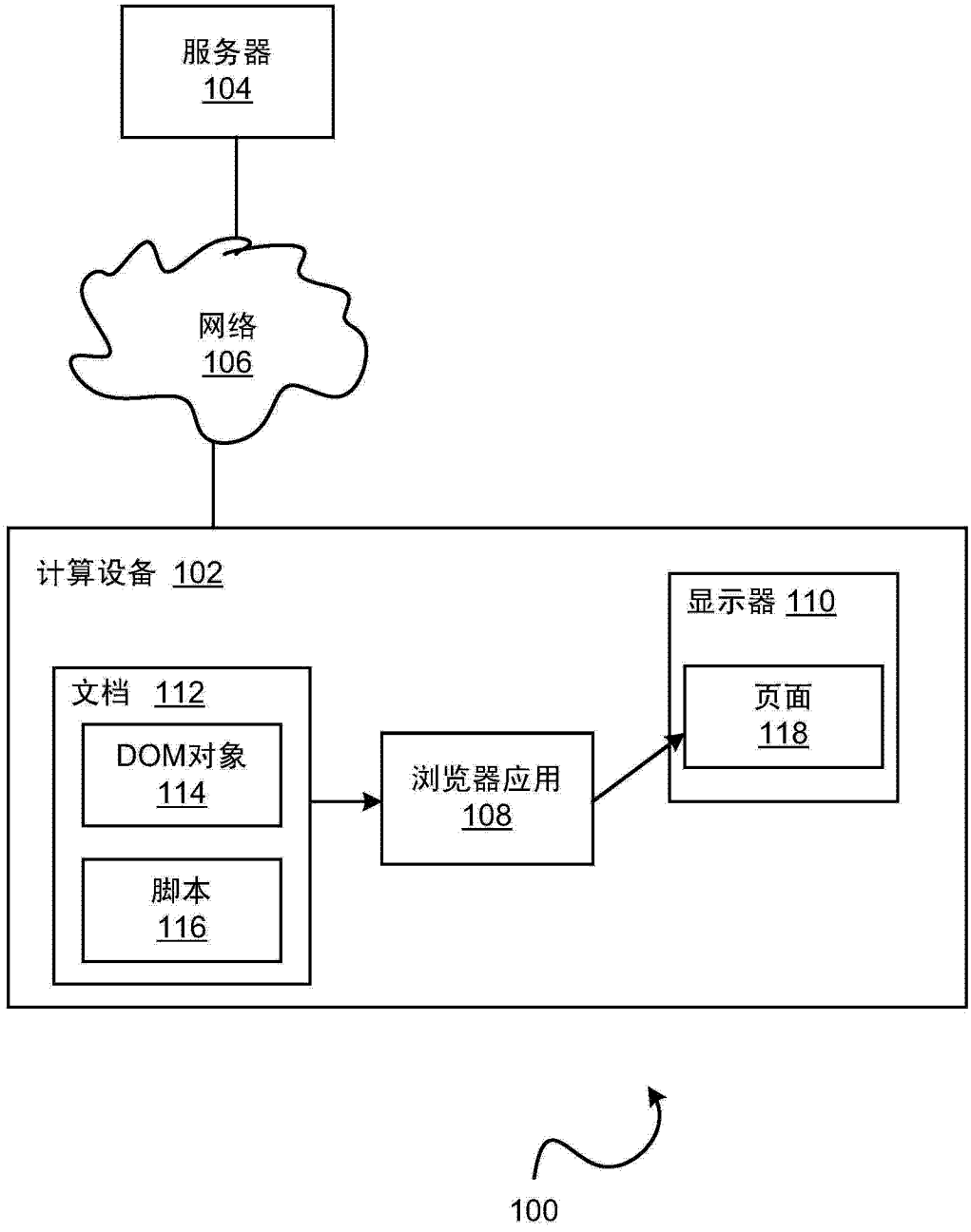

服务器
104
网络
106
计算设备 102
显示器 110
文档 112
DOM对象
114
脚本
116
浏览器应用
108
页面
118
100

图 1a

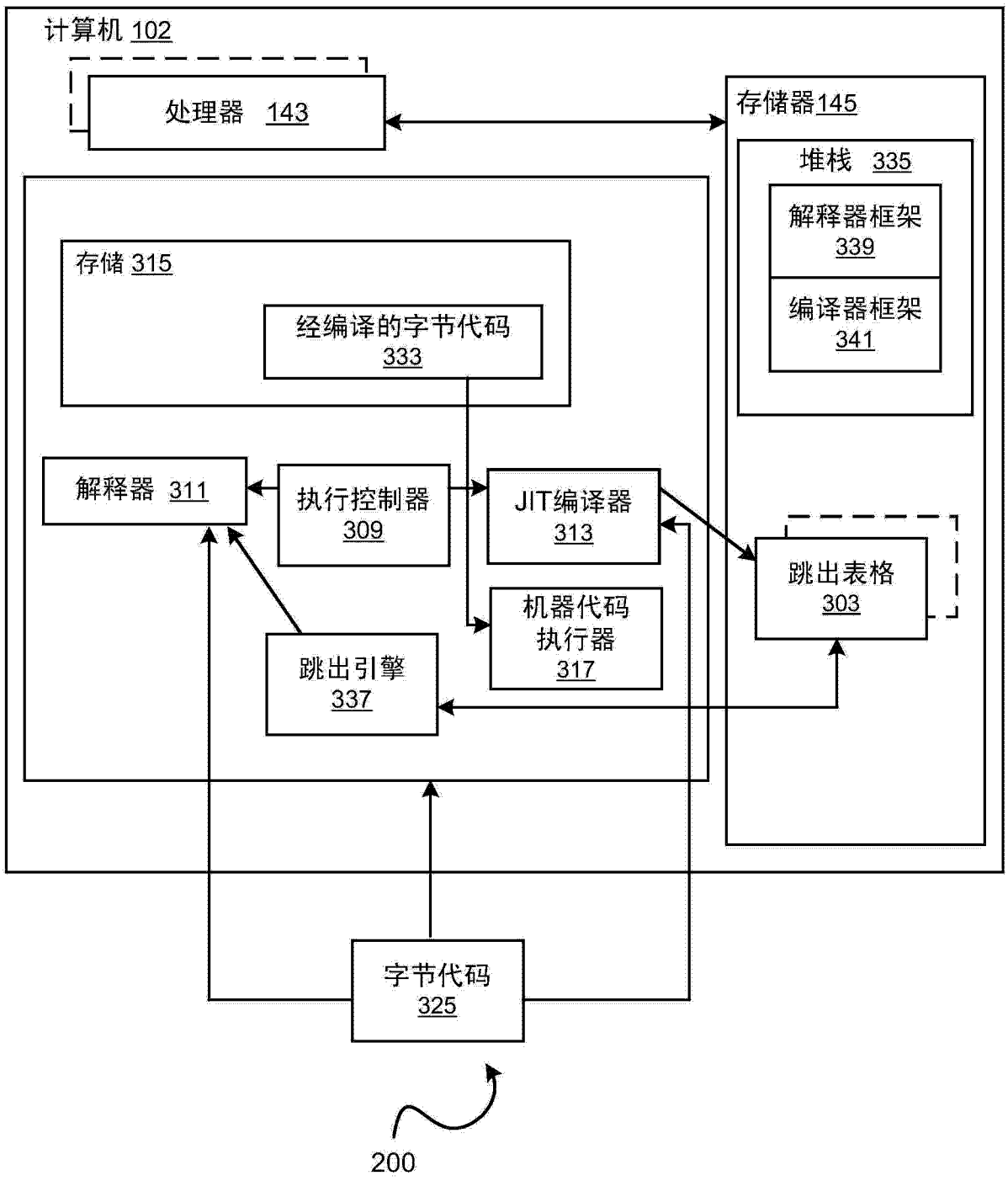

计算机 102
处理器 143
存储器145
堆栈 335
解释器框架 339
编译器框架 341
存储 315
经编译的字节代码 333
解释器 311
执行控制器 309
JIT编译器 313
跳出表格 303
机器代码执行器 317
跳出引擎 337
字节代码 325
200

图 1b

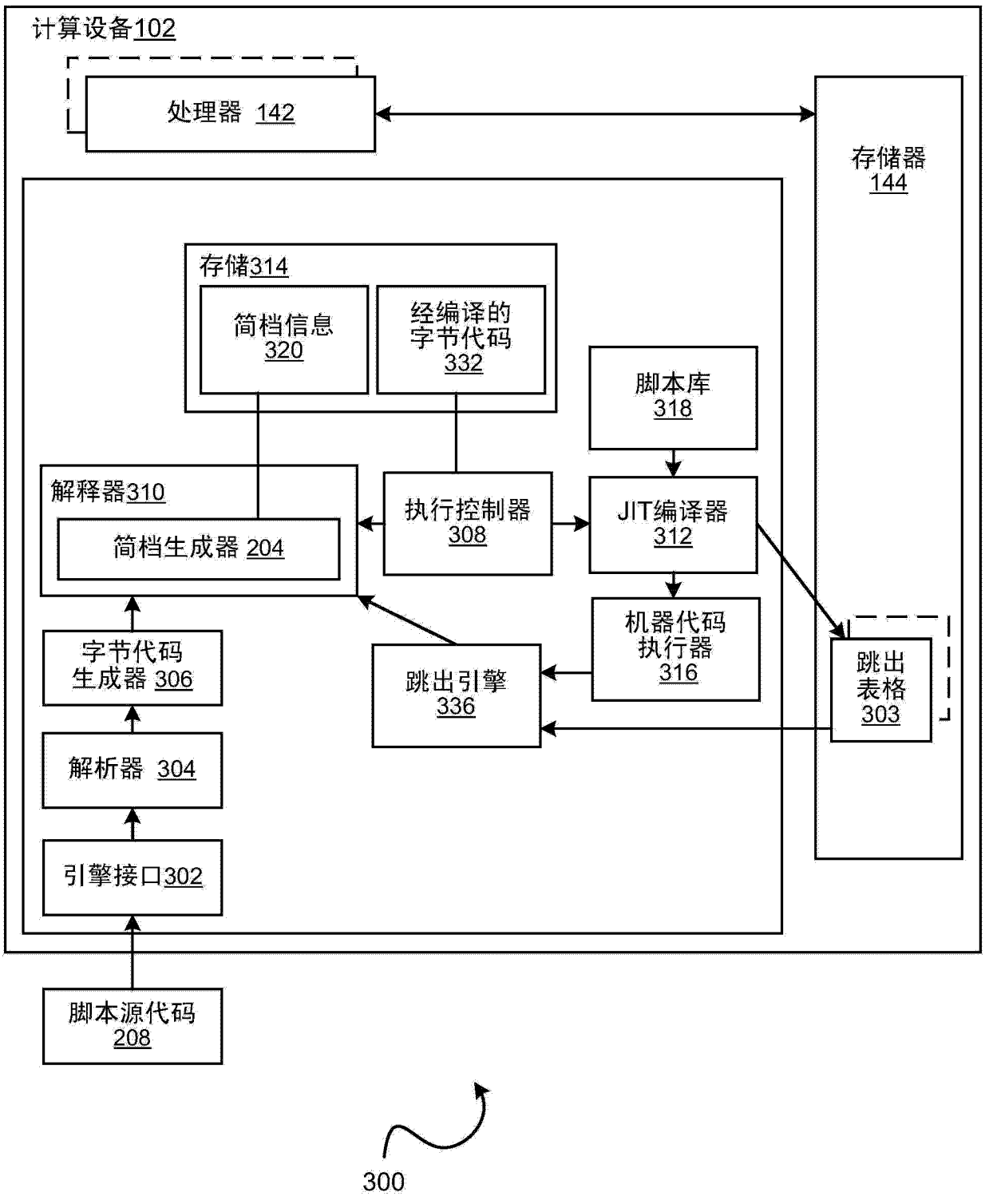

计算设备102
处理器 142
存储器
144
存储314
简档信息
320
经编译的
字节代码
332
脚本库
318
解释器310
简档生成器 204
执行控制器
308
JIT编译器
312
机器代码
执行器
316
跳出
表格
303
字节代码
生成器 306
跳出引擎
336
解析器 304
引擎接口302
脚本源代码
208
300

图 1c

JavaScript

```
function foo(a, b)
{
   return a + b;
}
```
152

字节代码

```
t1 = ArgIn a          line #1
t2 = ArgIn b          line #2
t0 = Add t1, t2       line #3
return t0             line #4
```
154

所生成的代码

```
reg1 = a                  line #1
if (reg1 is not an int)   line #2
        bailout()         line #3   → 在行#2处的跳出点1，t1在reg1中，t2和t0不是活的
reg2 = b                  line #4
if (reg2 is not an int)   line #5
        bailout()         line #6   → 在行#3处的跳出点2，t1在reg1中，t2在reg2中，t0不是活的
reg3 = Add reg1, reg2     line #7
if (add overflowed)       line #8
        bailout()         line #9   → 在行#3处的跳出点3，t1在reg1中，t2在reg2中，t0不是活的
return reg3               line #10
```
156

在所生成的代码156的行#3处的跳出点1的表格1 156

行#2 t1在reg1中，t2是不活动的，t0是不活动的	158

在所生成的代码156的行#6处的跳出点2的表格2 156

行#3 t1在reg1中，t2在reg2中，t0是不活动的	160

在所生成的代码156的行#9处的跳出点3的表格3 156

行#3 t1在reg1中，t2在reg2中，t0是不活动的	162

150

图 1d

所生成的代码(浮点类型专用化)

```
reg1 = a                                          line #1
if (reg1 is not a JavaScriptNumber)               line #2
        bailout()                                 line #3  → 在行#2处的跳出，t1在reg1中，t2/t0不是活的
floatReg1 = reg1->floatValue                      line #4
reg1 = b                                          line #5
if (reg1 is not an JavaScriptNumber)              line #6
        bailout()                                 line #7→ 在行#3处的跳出，t1在floatReg1（值是类型专用
                                                           化的浮点）中，t2在reg1中，t0不是活的

floatReg2 = reg1->floatValue                      line #8
floatReg3 = floatAdd floatReg1, floatReg2                 line #9
reg1 = ConstructJavascriptNumber(floatReg3)               line #10
return reg1                                       line #11
```

170

图 1e

JavaScript

函数bar()

```
{
   return 10;                    172
}
```

函数foo(a, b)

```
{
        var c = a + b;
        if (bar() != 10)
        {
                return c;
        }

        return 0;
}                                 173
```

优化的所生成的代码

```
reg1 =a                          line #1
if (reg1 is not int)             line #2                                    174
        bailout()                line #3
reg2 = b                         line #4
if (reg2 is not int)             line #5
        bailout()                line #6
reg3 = reg1 + reg2               line #7

if (bar is not “function bar”)   line #8
        bailout()                line #9     ->保持reg3活的，因为解释器将需要'c'。
return 0                         line #10
```

171

图 1f

JavaScript

180

```
function bar(a, func)
{
        return func(a, 100) + a;
}
```
182

```
function foo(a, b)
{
        return a + b;
}
```
183

```
bar(2, foo)
```
184

字节代码

```
function bar:
t1 = a                  line #1
t2 = 100                line #2
t3 = func               line #3
t4 = call t3(t1, t2)    line #4
t0 = Add(t4, t1)        line #5
return t0               line #6
```
185

所生成的代码(将foo内联到bar):

```
reg1 = a
reg3 = func
if (reg3 is not foo)
        bailout()       ->到bar #4的跳出，t1在reg1中，t2是100，t3在reg3中，t4/t0不是活的
```
186

```
if (reg1 is not int)
        bailout()       ->到foo #2的跳出，t1在reg1中，t2/t0不是活的，随后到bar #5的跳出，
                          t1在reg1中，t4是从到foo的跳出的解释器返回值，t0/t2/t3不是活的
```
188

```
reg0 = Add reg1, 100
if (add overflowed)
        bailout()       ->到foo #3的跳出，t2是100，t0不是活的，随后到bar #5的跳出，
                          t1在reg1中，t4是从到foo的跳出的解释器返回值，t0/t2/t3不是活的
```
189

```
reg0=Add reg0, reg1
If (add overflower)
        bailout()       ->  到bar #5的跳出，t1在reg1中，t4在reg0中，t0/t2/t3不是活的
return reg0
```

图 1g

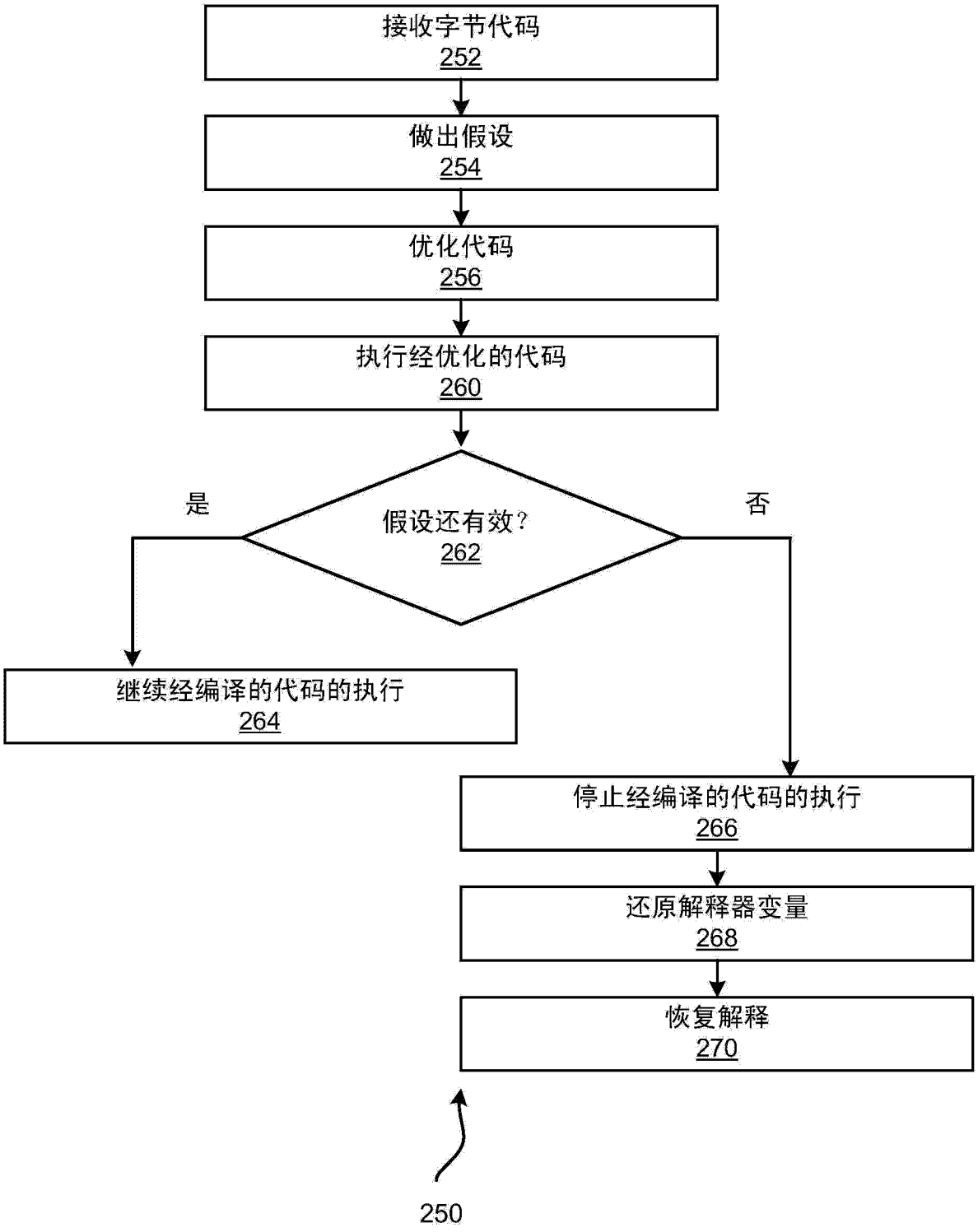

接收字节代码
252
做出假设
254
优化代码
256
执行经优化的代码
260
假设还有效？
262
是
否
继续经编译的代码的执行
264
停止经编译的代码的执行
266
还原解释器变量
268
恢复解释
270
250

图 2

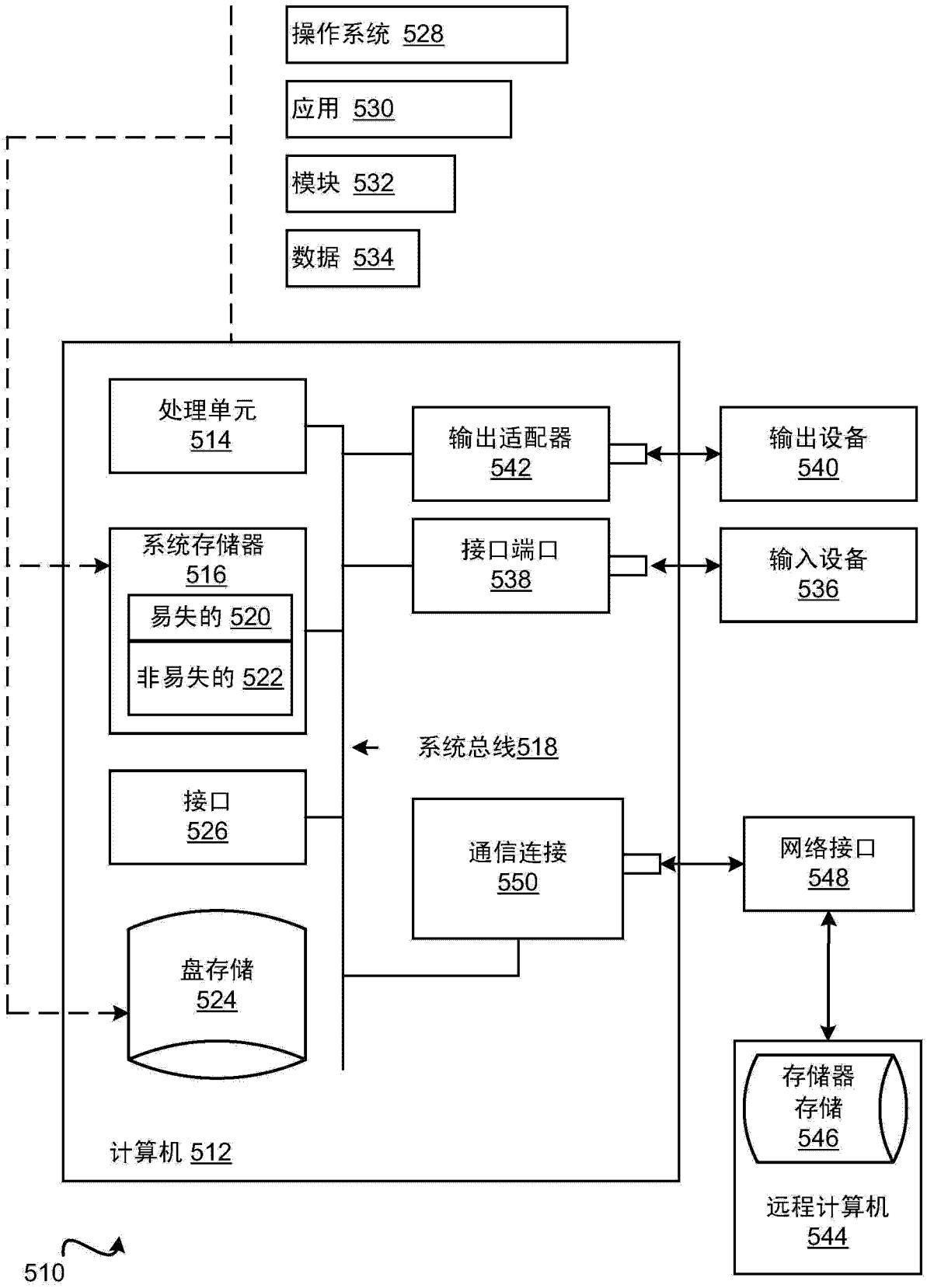

操作系统 528
应用 530
模块 532
数据 534
处理单元 514
系统存储器 516
易失的 520
非易失的 522
接口 526
盘存储 524
输出适配器 542
接口端口 538
系统总线518
通信连接 550
输出设备 540
输入设备 536
网络接口 548
存储器存储 546
远程计算机 544
计算机 512
510

图 3