



## [12] 发明专利申请公开说明书

[21] 申请号 200410085101.X

[43] 公开日 2005 年 9 月 7 日

[11] 公开号 CN 1664779A

[22] 申请日 2004.6.25

M · L · 罗伯茨 M · R · 普莱斯克

[21] 申请号 200410085101.X

V · K · 格罗弗

[30] 优先权

[74] 专利代理机构 上海专利商标事务所有限公司

[32] 2003. 7. 25 [33] US [31] 10/628,054

代理人 谢喜堂

[32] 2003. 6. 27 [33] US [31] 10/607,591

[32] 2003. 6. 30 [33] US [31] 10/610,692

[32] 2003. 7. 23 [33] US [31] 10/626,251

[32] 2003. 7. 22 [33] US [31] 10/625,892

[32] 2003. 6. 26 [33] US [31] 10/609,275

[71] 申请人 微软公司

地址 美国华盛顿州

[72] 发明人 A · V · S · 萨斯特里

A · E · 艾尔斯 A · J · 爱德华兹

C · L · 米谢尔 D · M · 吉尔斯

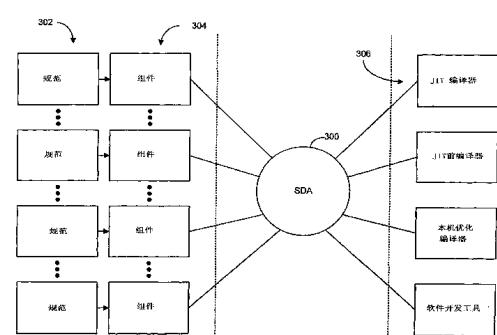
D · R · 小塔迪蒂 J · 伯格

权利要求书 5 页 说明书 34 页 附图 27 页

[54] 发明名称 软件开发基础架构

[57] 摘要

提供了一种软件开发体系结构，用于结构各类的软件开发工具。能够通过将指定了专用于一组软件开发场景的功能的规范集成到与软件开发场景无关的框架里来创建软件开发工具。然后能够编译集成的规范，以创建软件开发工具。替换地，也可以在不访问源代码的情况下，在运行时实现集成。所述体系结构能够使用下列内容的任何组合：软件场景独立的中间表示格式，能够支持多个程序设计语言专用异常处理模型的一个或多个异常处理模型，能够表示多个源语言的类型表示的类型系统，以及能够生成以多个执行体系结构为目标的代码的代码生成器。



1.具有用于实现软件开发体系结构的计算机可执行指令的一个或多个计算机可读介质，其特征在于，该软件开发体系结构包括：

5       一软件开发场景不相关的中间表示格式；

一个或多个异常处理模型，用于支持多个程序设计语言专用异常处理模型；

一类型系统，用于表示多个源语言的类型表示；以及

一代码生成器，用于生成以多个执行体系结构为目标的代码。

10      2.如权利要求1所述的一个或多个计算机可读介质，其特征在于，所述体系结构可调节以产生范围从轻型的JIT编译器到整体程序优化编译器的目标软件开发工具。

15      3.如权利要求1所述的一个或多个计算机可读介质，其特征在于，所述体系结构能够被配置成产生具有不同范围的存储器占用面积、编译速度和优化的  
目标软件开发工具。

4.如权利要求1所述的一个或多个计算机可读介质，其特征在于，所述软件开发体系结构可用于产生可通过将修改组件与该软件开发体系结构相组合来进行修改的软件开发工具。

20      5.如权利要求1所述的一个或多个计算机可读介质，其特征在于，所述软件开发体系结构可用于通过将软件开发体系结构的二进制版本与修改组件动态地进行链接来产生软件开发工具。

6.如权利要求1所述的一个或多个计算机可读介质，其特征在于，所述中间表示格式在采用该中间表示格式的软件工具的运行时是可扩展的。

25      7.如权利要求1所述的一个或多个计算机可读介质，其特征在于，所述体系结构可与一个或多个软件开发组件相组合。

8.如权利要求7所述的一个或多个计算机可读介质，其特征在于，所述一个或多个软件开发组件包括描述目标软件开发工具的数据。

9.如权利要求7所述的一个或多个计算机可读介质，其特征在于，所述一个或多个软件开发组件为所述代码生成器提供了目标执行体系结构数据。

30      10.如权利要求7所述的一个或多个计算机可读介质，其特征在于，所述一

一个或多个软件开发组件为所述类型系统提供了一个或多个类型校验规则。

11.如权利要求7所述的一个或多个计算机可读介质，其特征在于，所述一个或多个软件开发组件为所述体系结构提供了一组类扩展声明。

12.如权利要求7所述的一个或多个计算机可读介质，其特征在于，所述组合的一个或多个软件开发组件和体系结构产生一目标软件开发工具。  
5

13.如权利要求12所述的一个或多个计算机可读介质，其特征在于，所述目标软件开发工具包括本机编译器。

14.如权利要求12所述的一个或多个计算机可读介质，其特征在于，所述目标软件开发工具包括JIT编译器。

10 15.一种创建目标软件开发工具的方法，其特征在于，所述方法包括：

接收指定了专用于一个或多个软件开发场景的功能的至少一个计算机可读规范；

根据所述至少一个规范创建至少一个软件开发组件；以及  
将所述至少一个软件开发组件集成到一软件开发场景不相关框架中。

15 16.如权利要求15所述的方法，其特征在于，它还包括：

编译所述至少一个软件开发组件和框架，以创建所述目标软件开发工具。

17.如权利要求15所述的方法，其特征在于，根据多个计算机可读规范，  
为多个相应的软件开发场景创建的软件开发组件被集成到所述框架中。

18.如权利要求17所述的方法，其特征在于，所述多个计算机可读规范为  
20 以下相应的软件开发场景指定了功能：

目标执行体系结构；  
输入语言或者输入二进制格式；以及  
编译类型。

19.如权利要求15所述的方法，其特征在于，所述计算机可读规范为所述  
25 软件开发工具的目标执行体系结构指定了功能。

20.如权利要求15所述的方法，其特征在于，所述计算机可读规范为所述  
软件开发工具指定了用于适应输入语言的功能。

21.如权利要求15所述的方法，其特征在于，所述计算机可读规范为所述  
软件开发工具指定了用于适应二进制输入的功能。

30 22.如权利要求15所述的方法，其特征在于，所述计算机可读规范包括用

于类型校验一个或多个语言的一个或多个规则组。

23. 如权利要求15所述的方法，其特征在于，所述计算机可读规范包括专用于一个或多个软件开发场景的一组类扩展声明。

24. 如权利要求15所述的方法，其特征在于，所述计算机可读规范包括用  
5 于处理能够表示多个程序设计语言的中间表示格式的功能。

25. 如权利要求24所述的方法，其特征在于，所述中间表示格式包括能够支持多个程序设计语言专用异常处理模型的一个或多个异常处理模型。

26. 如权利要求24所述的方法，其特征在于，所述中间表示包括能够表示多个程序设计语言的类型表示的类型表示。

10 27. 如权利要求15所述的方法，其特征在于，它还包括：  
集成专用于所述软件开发场景之一的自定义码。

28. 如权利要求15所述的方法，其特征在于，所述软件开发工具包括以下组中的一个：本机编译器、JIT编译器、分析工具和CDK。

15 29. 如权利要求15所述的方法，其特征在于，所述计算机可读规范指定了以下组中的一个的功能：预JIT编译器功能、编译器功能、优化器功能、缺陷检测工具功能。

30. 包含用于执行如权利要求15所述的方法的一个或多个计算机可执行指令的一个或多个计算机可读介质。

31. 一种用于从一公用框架创建目标软件开发工具的方法，其特征在于，  
20 该方法包括：

基于所述目标软件开发工具的一个或者多个特征配置所述公用框架；  
将包括所述目标软件开发工具的一个或多个特征的数据集成到所述公用框架中；

从所述集成的公用框架创建所述目标软件开发工具。

25 32. 如权利要求31所述的方法，其特征在于，所述一个或多个特征可包括所述目标软件开发工具在目标体系结构上执行所必需的存储量、所述目标软件开发工具在目标体系结构上执行的速度、所述目标软件开发工具的输入语言、所述目标软件开发工具的输入二进制格式、或者所述目标软件开发工具用于在目标体系结构上执行的目标体系结构。

30 33. 一种创建目标软件开发工具的方法，其特征在于，所述方法包括：

- 接收至少一个计算机可读规范，该计算机可读规范指定：
- 待创建的软件开发工具的类型，  
用于所述软件开发工具的目标执行体系结构的功能，  
用于适应所述软件开发工具的输入语言的功能，以及  
5 用于处理能够表示多个程序设计语言的中间表示的功能，  
其中，所述至少一个计算机可读规范包括一组类扩展声明；  
将所述至少一个计算机可读规范集成到软件开发体系结构中；以及  
经由所述集成的规范和体系结构创建所述软件开发工具，其中，所述软件  
开发工具具有指定的类型、其目标为指定的目标执行体系结构、适应指定的输  
10 入语言、并且处理所述中间表示。
- 34.一种产生内部兼容软件开发工具的方法，其特征在于，所述方法包括：  
从一软件开发体系结构创建第一软件开发工具；以及  
基于所述第一软件开发工具创建第二软件开发工具，其中，所述第二软件  
开发工具被动态地链接到所述软件开发体系结构的二进制版本。
- 15 35.如权利要求34所述的方法，其特征在于，所述软件开发体系结构的二  
进制版本包含可经由一组声明来扩展的类。
- 36.如权利要求34所述的方法，其特征在于，所述软件开发体系结构包括  
用于由所述第一和第二软件开发工具都使用的中间表示的功能。
- 37.如权利要求34所述的方法，其特征在于，所述软件开发体系结构包括  
20 用于由所述第一和第二软件开发工具都使用的类型系统的功能。
- 38.如权利要求34所述的方法，其特征在于，所述软件开发体系结构包括  
用于由所述第一和第二软件开发工具都使用的异常处理模型的功能。
- 39.一种修改软件开发工具的方法，所述软件开发工具是使用包括一个或  
多个软件开发组件的软件开发体系结构所创建的，其特征在于，所述方法包括：  
25 将所述软件开发体系结构中不存在的软件开发组件动态地链接至所述软件  
开发体系结构的二进制版本；以及  
根据所述动态链接的二进制版本和所述软件开发组件创建一修改的软件开  
发工具。
- 40.如权利要求39所述的方法，其特征在于，所述软件开发体系结构的二  
30 进制版本包括可经由一组声明来扩展的类。

41.如权利要求39所述的方法，其特征在于，所述软件开发体系结构的二进制版本包括用于由所述修改的软件开发工具使用的类型系统的功能。

42.如权利要求39所述的方法，其特征在于，所述软件开发体系结构的二进制版本包括用于由所述修改的软件开发工具使用的异常处理模型的功能。

5 43.一种创建软件开发工具的方法，其特征在于，所述方法包括：

接收至少一个计算机可执行文件，所述计算机可执行文件包括：

一中间表示，它能够表示多个程序设计语言和计算机可执行图像；

一个或多个异常处理模型，它能够支持多个程序设计语言专用异常处理模型；

10 一类型系统，它能够表示多个源语言的类型表示；以及

一代码生成器，它能够生成以多个执行体系结构为目标的代码；

使用至少一个类扩展声明将一软件组件与至少一个计算机可执行文件链接；以及

经由所述链接的软件组件和计算机可执行文件创建所述软件开发工具。

## 软件开发基础架构

5 相关申请数据本申请是以下美国专利申请的部分延续申请，并要求这些申请的优先权，所有这些申请在此被引入以供参考：2003年6月27日提交的、发明人为Mark Ronald Plesko和 David Read Tarditi, Jr.的、名称为“TYPE SYSTEM FOR REPRESENTING AND CHECKING CONSISTENCY OF HETEROGENEOUS PROGRAM COMPONENTS DURING THE PROCESS OF 10 COMPILATION”的第10/607,591号申请；2003年6月30日提交的、发明人为Julian Burger、代理人参考号码第3382 - 64708号、名称为“GENERATING SOFTWARE DEVELOPMENT TOOLS VIA TARGET ARCHITECTURE SPECIFICATION”的第10/610,692号申请；2003年7月23日提交的、发明人为Julian Burger、David Read Tarditi,Jr.、Charles L. Mitchell、Andrew Edward Ayers和Vinod K. Grover的、 15 代理人参考号码第3382 - 65690号、名称为“DESCRIPTION LANGUAGE FOR AN EXTENSIBLE COMPILER AND TOOLS INFRASTRUCTURE”的第10/626,251号申请；2003年7月22日提交的、发明人为Charles L. Mitchell、Julian Burger、Vinod K. Grover和David Read Tarditi, Jr.、代理人参考号码第3382 - 65679号、名称为“GENERAL PURPOSE INTERMEDIATE REPRESENTATION OF 20 SOFTWARE FOR SOFTWARE DEVELOPMENT TOOLS”的第10/625,892号申请；以及2003年6月26日提交的、发明人为Vinod K. Grover和Akella V.S. Sastry、代理人参考号码第3382 - 65591号、名称为“AN INTERMEDIATE REPRESENTATION FOR MULTIPLE EXCEPTION HANDLING MODELS”的第10/609,275号申请。

25 技术领域

本技术领域涉及软件开发，具体来讲涉及一种用于帮助生成软件开发工具的体系结构。

背景

可使用多种程序设计语言，以便在编程时为程序员提供每种语言所特有的 30 有益之处。类似地，可以使用多种处理器，以便在执行特定工作时提供每种

处理器所特有的有益之处。例如，嵌入式处理机特别适合于处理电子设备内部的意义明确的任务，而通用处理机诸如Intel®Pentium®处理器更加灵活，并且能够处理复杂任务。此外，还存在为帮助程序员解决对于软件可靠性、安全性和高性能的逐渐增长的需求而创建的各种工具类型。因此，计算环境、结构配置和设备器件方面的多样性正在增加。因此，软件开发者面临在大量眼花缭乱的不同软件开发场景中进行适应和工作。

解决这种多样性的需求使得已经极其复杂的积木式软件开发工具的领域变得更加复杂。这样的软件开发工具可能包括各种组件，诸如编译程序，反编译程序，解码器，编码器，指令选择组件，以及指令认可组件。往往，这样的组件已经叠加覆盖了要求，并且组件本身可能在超过一个开发工具中出现（例如，在编译器和调试器中）。

尽管普遍需要类似的组件功能，但是难以开发出共享设计和实现方式的组件，特别是当涉及大量的程序设计语言或者其他程序表示时。往往，每一工具的每一个组件具有它自己的实现方式，这导致大量冗余的工作和重复的代码。另外，在体系结构之间、甚至在相同的源库内部几乎不具有一致性。因此，对一个组件的代码作出任何改进对于一种特定实现方式都是有用的，但是该改进不会自动地扩展至实现同一功能的其他工具或者体系结构。最终，在解决一特定软件开发场景方面付出的努力通常必须被重复进行，以解决不同的场景。

## 20 发明概述

提供了一种软件开发体系结构( SDA)，用于结构软件开发工具的各类组件。这些组件可被用于以一种或多种源语言编写的程序或者诸如二进制可执行文件之类的计算机可读输入。然后这些组件可被合并，以创建软件开发工具。所述SDA包括各个方面。所述各个方面可被单独地和独立地使用，或者所述各个方面可以不同组合和子组合的方式来使用。

在一方面，SDA使用能够表示多种程序设计语言及其他诸如二进制可执行文件的计算机可读程序格式的中间表示，一个或多个能够支持多种程序设计语言或者其他计算机可读程序格式的异常处理模型，以及能够表示多种源语言或者其他计算机可读程序格式的类型表示的类型系统。此外，所述中间表示能够表示可为多种执行体系结构执行的二进制码。因此，使用这种中间表示编写

的组件可以被应用于以各种程序设计语言编写的程序、诸如二进制可执行文件或者目标文件之类的程序的计算机可读表示、以及用于特定目标结构的程序。这通过允许共享组件降低了开发软件工具的成本。它有助于改进对由不同种类的组件所组成的程序的分析和优化。 组件能够包括数据流分析，控制流分析，  
5 程序转换，数据表示优化，寄存器分配程序，以及指令调度程序。

在一个方面中，提供了一种方法，用于通过将用于实现专用于软件开发场景的功能集成到与软件开发场景无关的体系里，来创建软件开发工具的组件。 这样的软件开发场景可能涉及程序设计语言，目标执行体系结构，中间表示级，等等。

10 在又一个方面中，提供了计算机可执行软件，用于产生由中间表示和共享组件的扩展版本所组成的软件开发工具。 所述软件接受对于多种描述目标软件开发工具的软件的多种配置之一的选择；能够将专用于所述目标软件开发工具的数据合并到中间表示里；并且能够产生包括符合所述配置和数据的目标软件开发工具的组件。 以这种方式，组件和中间表示能够被扩展到在新的和未  
15 预见到的情况下使用，例如对于现有工具的新要求、新颖类型的工具、新的或者改进的程序设计语言以及新的计算机体系结构。

这些及其他方面将根据以下参照附图所作出的详细说明而变得更加清楚。

#### 附图说明

图1示出用于结构各类软件开发工具的SDA的方框图。

20 图2(a)是使用SDA创建软件开发工具的方法的流程图。

图2(b)示出使用SDA创建软件开发工具的方法的另一个流程图。

图3是用于使用SDA创建软件开发工具的组件的系统的高级方框图。

图4示出用于使用诸如图3中所示出的系统创建的三个目标执行体系结构中每一个的编译器、调试器和优化器的框图。

25 图5示出在使用SDA所创建的编译器中的示例性的编译过程的方框图。

图6(a)-(d)贯穿了从在源中读取、到高级的与机器无关的IR、到低级的与机器有关的IR的IR转换。

图7是用于在编译的各阶段对IR进行类型校验的编译器系统的一个实施例的方框图。

30 图8是一种供IR使用的类型检验器的方框图。

- 图9图示出用于实现IR中的统一异常处理的系统。
- 图10A图示出使用IL阅读程序产生异常处理结构的中间表示的统一集合的方法。图10B示出用于根据软件的统一IR生成可执行的方法。
- 图11是一种用于在以多个IL表示的形式表达的多个源语言内部产生异常处  
5 理结构的简单的和统一的IR的系统的方框图。
- 图12是用于代码生成的系统的方框图。
- 图13是一种以软件实现的用于代码生成的方法的实施例的流程图。
- 图14A描述了一种用于扩展核心类定义以便通过扩展核心框架来构建工具的全过程。  
10 图14B描述了用于通过使用与软件场景有关的扩展来扩展SDA核心框架以便构建软件开发工具的全过程。
- 图15A图示出一种用来将扩展添加到核心编译器框架以便扩展它的方法。
- 图15B图示出一种用来将核心编译器框架编译为与扩展独立的文件的方法。  
15 图16图示出一种用于在编译时间之前静态地扩展与核心框架程序相关的类的方法。
- 图17图示出用于实现图16中的过程的示例性系统。
- 图18图示出用于通过在运行时将扩展链接至适当的核心类来扩展可扩展核心框架软件程序的核心类定义的方法。  
20 图19图示出用于实现图18中的过程的示例性系统。
- 图20是用于可交付使用的计算机软件的系统的方框图。
- 图21是示出图20中所描述的可执行软件的具体范例的方框图。
- 图22示出可交付使用的计算机软件的另一个实施例。
- 图23示出用于修改先有软件开发工具的可交付使用的计算机软件。  
25 图24图示出根据SDA创建相互兼容的软件工具。
- 图25图示出作为SDA的实施例的操作环境使用的计算机系统的范例。
- 详细说明
- 提供了一种SDA，用于结构用于各类软件开发工具的组件、并且连接这些组件以形成工具。该SDA能够结构各种软件开发工具，所述软件开发工具可以  
30 采用任何数目的程序设计语言作为用于任何数目的目标执行体系结构的输入和

输出代码。

#### 示例性的目标执行体系结构

此处所述的目标执行体系结构能够包括各种硬件机器或者虚拟计算机中的任何一种。 目标执行体系结构能够包括用于执行代码的任何环境。 这样的目标执行体系结构能够包括Intel®x86、AMD、IPF、ARM和MIPS体系结构，以及其他体系结构，包括将来实现的那些体系结构。

Intel®x86体系结构包括、但是不局限于：可以从Intel公司获得的基于Intel®x86体系结构的任何处理器，诸如80x86, 80x88, Intel86, Intel286, Intel386, Intel486, 和Pentium处理器。 AMD体系结构包括、但是不局限于：可以从Advanced Micro Devices ( AMD)公司获得的AMD64和AMD32体系结构。 IPF (Itanium 处理器系列)体系结构包括、但是不局限于可以从Intel公司获得的IA64体系结构。 ARM体系结构包括可以从ARM有限公司获得的大量16和32位嵌入式RISC微型处理器。 MIPS体系结构包括、但是不局限于可以从MIPS Technologies公司获得的MIPS64™和MIPS32™体系结构。

#### 15 示例性的软件开发工具

此处所述的软件开发工具能够包括各种可用于开发软件的工具中的任何一种。 这样的工具可以包括本机码编译器，即时( JIT ) 编译器，调试器，模拟器，分析工具，缺陷检测工具，编译器开发工具包( CDK )和优化器。

这样的软件开发工具能够包括诸如程序转换组件、程序分析组件、编译程序，反编译程序，解码器，编码器，指令选择组件，以及指令合法化 (legalization) 组件等等之类的组件在某些情况下，组件自身能够用作工具。

#### 示例性的软件开发场景(scenario)

大量软件开发场景中的任何一种均能够影响SDA的功能。 例如，一种用于特定软件开发工具的软件开发场景可能包括所述软件开发工具将以之为目标的各种目标执行体系结构（例如，IPF, X86, AMD, ARM等等）。同时，软件开发场景可能与一种正被执行的编译相关（例如，JIT或者本机优化编译器）。 软件开发场景还可能与软件开发工具所执行的其他类型功能相关，诸如类型分析、优化、模拟、调试、代码生成等等。 另外一种软件开发场景还可能与一种特定程序设计语言（例如，JAVA, C++, C#等等）相关，所述软件开发场景是为该特定程序设计语言而被专门配置的。 这样的语言可能具有不同的异常

处理模型。此外，软件开发场景还可能涉及该工具是否被用于被管理的执行环境（例如，由Microsoft.NET框架提供的Microsoft CLR的环境）。其他场景也是可行的。

- SDA可被用于为解决任何一种或多种软件开发场景的工具创建组件。  
5 例如，SDA可被用于为可操作的接受任何一种或多种程序设计语言、生成多种目标执行体系结构中任何一种等等的代码的工具创建组件。

### 软件开发体系结构

图1示出一种SDA的方框图，该SDA被用作使用该SDA结构采用多种配置中的任何一种中的任何数目的组件、来结构各类软件开发工具的基础。组件集合100表示能被任何配置使用的模块。这些模块是使用由SDA提供的核心功能来构建的。SDA核心100可以作为目标文件或者源代码的集合来提供。SDA核心100提供了多个被管理的和/或本机的应用程序接口102 (API)。围绕SDA核心100和API 102的每一方框均表示可使用SDA构建的软件开发工具。这些工具可以包括本机编译器104，预JIT编译器106，JIT编译器108，优化工具110，  
10 缺陷检测工具112，分析工具114，和编译器开发工具包(CDK)116。  
15

本机编译器104表示用于多个目标执行体系结构和多种源语言的本机机器代码的一个或多个编译器。例如，使用SDA构建的本机编译器104可以将以C++程序设计语言编写的程序编译为用于在x86目标执行体系结构上执行的本机码。

20 预JIT编译器106能够在在任何数目的目标执行体系结构上运行应用程序之前，根据诸如用于微软的.NET平台的通用中间语言( CIL)之类的独立于体系结构的程序表示来优化和生成本机码。预JIT编译器能够在服务器上高速缓存地、在安装时间时、或者在载入时间（或者在后台中）在客户端上即时地运行。例如，SDA的预JIT用途106可以是用于创建为将CIL编译为在x86目标执行体系结构上执行的本机码而设计的预JIT编译器。  
25

JIT编译器108能够在运行时、按照需要在目标执行体系结构上根据诸如CIL之类的独立于体系结构的表示来编译代码（即时地）。例如，使用来自SDA的接口所构建的JIT编译器108可以是为在运行时、在AMD目标执行体系结构上将CIL编译为本机码而设计的。

30 缺陷检测工具110能够在以一种或多种语言编写的代码中（在程序运行之前）

静态地检测缺陷。 缺陷检测工具能够被结构，以便在任何数目的目标执行体系结构上运行。 例如，SDA的缺陷检测用途110可以是用于创建被设计为将以C++程序设计语言编写的代码作为输入并检测该代码中的缺陷的缺陷检测工具。 缺陷检测工具能够被设计为在例如xS6目标执行体系结构上执行。

5 分析工具112分析以一种或多种语言编写的代码。 分析工具能够被结构，以便在任何数目的目标执行体系结构上运行。 例如，SDA的分析工具用途112可以是用于创建被设计为采用以C#程序设计语言编写的代码、并确定哪些模块或者哪些行的源代码可能受到一个特定模块中的变化的影响的分析工具。 这被称为程序分片，并且在修改大系统的时候是很有帮助的。

10 优化器工具114优化以多种语言编写的代码。 例如，SDA的优化工具用途114可被用于创建被设计为采用以C++程序设计语言编写的代码、并基于配置文件优化字段布局的优化工具。 替换地，也可以创建一种优化工具，该优化工具被设计为优化来自诸如被管理的C++的任何微软.NET语言的CIL。

15 CDK用途116允许第三方独立地开发编译器。 CDK可以包括SDA的某些或者所有组件的二进制码、SDA的某些或者所有组件的库、以及用于某些或者所有组件的源代码，以便允许第三方修改该系统的特定方面。 例如，使用SDA构建的CDK工具116可以是由对迅速地并且成本有效地为使用WinCE的设备创建编译器感兴趣的芯片供应商所构建的。

20 所描述的工具使用所有或者一些SDA核心100元素。 另外，所描述的工具能够包括在SDA核心中不存在的另外的组件。 可以基于信息处理量、占用面积、宿主（Host）、目标以及代码质量要求，添加或者从配置中删除阶段和组件。

25 例如，本机编译器104可能具有在编译时间和存储器占用面积方面具有最高预算，并被期望产生最佳代码品质。 这种配置工具可能因此具有更多的代码优化。 对比起来，JIT编译器108可能要求更快速的编译时间和更小的存储器占用面积配置，该结构仍产生多少被优化的代码。 这种工具因此可能尽可能地维持与流程图、循环图表和SSA图表类似的数据结构，以避免费用高昂的重新结构。 另外，遍布该中间表示的路径的数目可以被最小化。

30 图2 (a)是使用SDA创建软件开发工具的方法的流程图。 框200示出由SDA接收的一个或多个规范。 在框202中，根据这些规范创建软件开发组件（例如，通过生成源代码并且随后编译）。 在框204中，组件被链接到SDA，以便创建

定制的软件开发工具。 替换地，提供另外的功能的定制代码也可以被添加给这些规范，并被集成到该组件中。 然后能够根据SDA和组件的组合创建软件开发工具。

5 替换地，图2(b)示出使用SDA的一种可执行的版本创建软件开发工具的方法的另一个流程图。 在框206处，提供了软件开发组件，并且在框208处，该组件被链接到SDA，以便创建定制的软件开发工具（例如，在运行时动态地创建，而不对源代码进行访问）

10 所述规范能够采用任何计算机可读语言或者适用于指定实现专用于一个或多个软件开发场景的功能的形式（例如，对象或者可执行的程序代码，源代码，或者定制语言）。

图3是用于使用SDA 300创建软件开发工具的组件的系统的高级方框图。 创建一个或多个规范302来描述软件开发场景的目标细节。 例如，规范302可能表明所述目标工具可以把哪一种程序设计语言作为输入，诸如Virtual Basic，C# 15 诸如CIL的.NET语言，或者C++。 同样地，规范可能表明用于目标软件开发工具的目标执行体系结构，诸如x86，AMD，MIPS，或者IPF体系结构。 另外， 规范可以包含其他信息，诸如用于对输入语言进行类型校验的规则集合，或者 20 用于扩展配置SDA核心数据结构的类扩展声明集合。 然而，规范302可被用于 创建能够被链接到SDA 300的软件开发组件304。 被链接的组件304和SDA 300 可以被组合使用，以便创建任何数目的工具306，诸如JIT编译器，预JIT编译器， 本机编译器，或者其他软件开发工具。 替换地，定制代码也可以被添加，以便 实现对于目标软件开发工具304的另外的功能。

因此，基于单一的源基础，诸如特定配置的SDA，能够通过简单地提供目标专用数据、以便为一种或多种软件开发场景定制目标软件开发工具，来创建任何数目的软件开发工具组件。 因此，能够考虑将软件开发工具划分为目标专用和目标不可知的模块两者。 图4图示出这一概念。 对于使用诸如图3中所示出的系统创建的三个目标执行体系结构中的每一个，图4中示出JIT、预JIT以及 Microsoft®Visual C++®本机编译器、调试器和优化器的框图。 因为用于执行体系结构的各种工具均是根据通用的SDA和规范构建的，所以由该规范提供的软件开发工具的目标专用码或者自定义码是根据能够由通用的SDA提供的目标不可知代码所划分的。 因此，用于目标执行体系结构的组件在不同类型软件开 30

发工具中能够是相同的，因为它们是根据相同规范创建的。

能够容易地在不同的执行体系结构之间对软件开发工具进行重定目标，因为仅仅需要改变目标专用组件，并且能够在大范围内修正目标专用或者目标不可知代码中找到的小缺陷。

5 例如，假定开发者希望对用于Intel xS6体系结构的JIT编译器400进行重定目标，以致它将在AMD或者MIPS体系结构上运行。开发者仅仅需要编写包括专用于AMD或者MIPS执行体系结构的数据在内的适当规范。然后，目标专用数据被集成到被用于创建Intel x86体系结构的JIT编译器400的相同或者类似的通用SDA中。然后，集成的SDA和规范被用于创建AMD体系结构的JIT编译器  
10 402，或者MIPS体系结构的JIT编译器404。

接下来，假定开发者在使用为Intel x86体系结构编写的规范结构的优化器406中发现软件缺陷。如果该软件缺陷是在目标不可知代码中找到的，则该代码很可能是通用SDA的一部分。因此，可能在图4中所示的整个工具之上出现该软件缺陷。

15 一旦改进是在使用通用的SDA构建的组件中或者通用的SDA自身中被开发，则能够通过在一个源代码库中实现修正，来同时发起对于所画出的十五个工具（对于三个不同的体系结构中的每一个的三个编译器、两个工具）所进行的改进。能够或者通过简单地使用更新的SDA为每一目标执行体系结构重新编译规范、或者通过为SDA和组件分配新的二进制码，来创建这十五个工具的  
20 校正版本。

类似地，如果在Intel®x86执行体系结构的优化器406的目标专用码中中找到软件缺陷，则该软件缺陷很可能是在为Intel®x86执行体系结构编写的规范中。此外，修正整个用于Intel®x86执行体系结构的工具之上的软件缺陷仅仅要求校正一个源代码（Intel®x86规范）并且使用通用的SDA重新编译更新的规范。

25 除了允许快速重定目标至替代的执行体系结构之外，图4中所示的软件开发工具还能够被快速地重新设计为接受另一种程序设计语言作为输入。例如，为了这一范例的目的，假定调试器408是根据一规范创建的，该规范表明调试器408采用C++作为输入，并且目标被定为用于MIPS执行体系结构。将调试器408重新设计为将C#作为输入、并且仍旧在MIPS体系结构上执行，是与重新编写  
30 规范以表明一种不同的输入语言同样简单的。新的规范能因此被集成到完全

相同的或者类似的SDA中，并且产生了一种采用C#作为输入的新的调试器。

### SDA核心数据结构和抽象

#### 机器模型—重定目标抽象

为了支持对使用此处所述的SDA创建的软件开发工具的快速重定目标，

- 5 SDA核心被划分为目标不可知和目标具体部分。只要可能，公用的目标专用码可被划分为具有到目标专用码的调用、以获取信息或者执行转换的目标不可知SDA。尽管仅仅是一个范例，但以下部分描述了SDA核心的一个实施例的实现细节。替代的SDA核心可以使用以下或者其他特征的任何组合。

#### 寄存器

- 10 能够经由API访问机器寄存器堆，来查询寄存器堆的数目（例如通用，浮点，预测），确定它们能够处理哪些机器类型（例如，int32, float64, MMX），以及获得表示机器寄存器的对象（例如，EAX, rO, p3）。寄存器对象能够从Layout（编排）类中来获得，并添加以下信息：数据流等等的内部计数数目；作为文本串的寄存器名；二进制编码；可扩展性对象。

15 机器相关操作码

能够经由API访问机器操作码表，以便查询操作码的数目，并且获得表示操作码的对象（例如，ADD, MOVZX）。操作码对象能够封装以下信息：内部操作码计数，作为文本串的操作码名称，属性，以及可扩展性对象。

#### 调用约定

- 20 能够通过降级阶段来显式地作出调用约定，并且能够经由目标不可知框架中的应用程序二进制接口（ABI）专用回叫来实现调用约定。该框架能够包括用于通用动作的子程序，诸如为寄存器分配参数，将寄存器操作码降级为存储器操作码（为了栈流通），等等。

- 25 能够通过使用描述在调用时使用的调用约定类型的目标专用属性来注释每一调用指令，来支持多个调用约定。这一属性能够被高级和独立于机器的优化阶段维护和传播。

#### 异常处理模型

- 30 能够与机器模型查询以及到任何所要求的目标专用码的回叫一起，目标不可知地实现各种异常处理( EH) 模型。将在下面的一个独立的部分中更详细地讨论异常处理。

## 代码生成

在这一实施例中，代码生成包含指令选择 / 降级，固有 / 切换 / 询问操作扩展，以及指令合法化。 寄存器分配和调度可以被视为独立的阶段。 将在下面的一个独立的部分中更详细地讨论代码生成。

### 5 编码

编码将编译器IR翻译为二进制机器代码的过程。 可以存在一种机器不可知框架，用于对所产生的目标专用码中出现的大量工作进行编码。

## 调度表

调度表可被用于全局和本地调度。 机器调度表可以包含每一机器指令所需要的指令等待时间和功能单元。 该表可以是由代码生成中所使用的重定目标工具生成的。

## 反编译程序

可以由代码生成文件驱动反编译。 它可以作为DLL来实现，所述DLL将使用公用接口由链接程序和调试器共享。

### 15 序言 / 收尾

运行序言程序和收尾函数可以在具有目标专用回叫的目标不可知框架中生成。 序言可以包含下列项中的任何或者所有：帧指针初始化；堆栈对界；堆栈分配；非易失性寄存器保存；异常处理结构初始化；运行时检查和初始化。 收尾可以包含下列项中的任何或者所有：非易失性寄存器恢复；堆栈解除分配；帧指针恢复；运行时检查。 每一次运行可能有两个帧指针：本地指针和参数指针。 这些可以是机器和功能相关的。

## 端（ENDIAN）支持

交叉定目标场景要求SDA知道端。 规范可以描述目标执行体系结构的端。 发射例程可以被设计为相应地遵守这一点。

### 25 堆栈分配

规范可以包含堆栈对界和生长方向（Growth Direction）信息。 堆栈分配包可以负责计算本地对准和所使用的总堆栈空间所要求的最大值。 另外，当不使用堆栈打包优化的时候，堆栈偏移量可以被分配给本地。

## 类型系统

30 IR的类型系统服务于许多重要的目的，并且将在下文中的一个独立的部分

中被更详细地讨论。

### 函数

函数或者方法对象可以描述组成单一过程的代码。以下属性可以在描述函数对象时被考虑：

- 5      1) 函数ID：方法的唯一标识符。
- 2) 函数名
- 3) 返回值类型
- 4) 调用约定
- 5) 函数签字或者自变量表
- 10     6) 异常信息

函数可以包含将被以非邻接的方式编排的规范。异常信息或者具有遵守函数编排的信息的任何其他数据结构可以被据此修改。

随着一个函数在各编译器阶段中往前进行，该函数可以是包含有对于后续阶段很关键的大部分信息的唯一实体。此外，该函数对象可以是一个经受大部分的转换、即代码分离、纳入、循环展开等等的函数对象。因此，与每一函数相关联的信息可以立即变为过时，除非有意识的保持让它随着转换一起移动。

为了让数据流分析在整个函数期间均是成功的，可取的是能表示一个整体的函数，而即使它不再是物理邻接的。

### 20    函数和方法

在本实施例中，可能无需区分函数和方法。这可能仅仅是在引入类层次结构的时候的术语分歧。

### 类

类对象可以是用于属于该类的所有函数或者方法的容器。以下属性可以25    描述类对象：

- 1) 类名
- 2) 大小
- 3) 函数表
- 4) 用于包括虚调用表的数据成员的类型 / 编排信息

30    由于在本实施例中，类定义可能不被分解为多个模块，所以类内部表示可

以为了优化目的而被重新排序是很重要的。一个这样的类重新排序的范例涉及由于存在与一个类对象有关的许多数据结构而进行的工作组缩减，其可能必须被重新排序，并且符合类内部编排。

### 阶段控制

5 由于被描述的SDA的可配置性和可扩展性，并且期望为连接组件提供统一的机制，可以经由数据而不是硬编码来提供阶段控制。阶段排序可以通过生成阶段对象表来指定。该阶段表可以随着配置、机器目标、以及用户可扩展性而变化。

10 每一阶段对象可以封装调试前置条件（将被执行的调试校验）、执行前置条件（控制阶段执行的编译标记，例如用于全局优化的-Od），阶段前可扩展性，阶段进入点，阶段后可扩展性，调试后置条件，文本阶段名，以及指向下一以及前一阶段的指针。

15 单独的阶段可以负责识别函数是“大的”并且适当地调节转换。例如，干扰包可以决定如果该函数具有超过n个的IR节点则放弃深入细致的分析，对确保校正持保守态度。或者优化可以决定对整个函数执行太过于昂贵，而是对区域运行，或者完全地将自身关闭。 ^

### 主驱动

20 主驱动负责初始化SDA以及任何连接的可扩展性模块。在起动时，可以依据编译器配置初始化各系统程序包（首先设置阶段表）。然后可以运行目标专用初始化（其可以提供可扩展性特征）。接下来，可以初始化可扩展性模块（其可以改变阶段表，属于各种数据结构，等等）。在此，可以调用命令行处理器。最终，如果没有发生错误，则可以控制编译管理（其可以是配置相关的）。

### 中间表示(IR)

25 在一个用于如此处所述地创建工具组件的SDA的实施例中，提供了单一的、通用的IR格式，用于在整个工具执行期间表示存储器中的用户程序。IR格式可以使用单一的、统一的形式表示从高级的、与机器无关的操作到低级的目标机器代码的代码范围。图5示出在使用SDA创建的编译器中的示例性的编译过程的方框图，所述SDA可以使用多种语言作为输入，提供多种语言作为输出，  
30 并且可将目标定为用于多种执行体系结构。

源代码500 - 506是以四种不同的源语言编写的。例如，源代码500是以C #编写的，而源代码506是以C++编写的。另外，PE二进制码及其他可执行的格式也可以被当作输入。首先处理源代码，并经由阅读程序508将其输入系统。然后，源语言被译成高级的IR (HIR)。然后可以在块510处，选择性地分析和优化HIR。HIR被翻译为中级电平IR (MIR)。这一表示低于HIR，但是仍是与机器无关的。在此，如在块512处所示，可以选择性地分析和优化MIR。然后，MIR在块514被代码生成译成与机器相关的低级的IR (LIR)。然后可以在块516处选择性地分析和优化LIR，并将其提供给块518处的发射器。发射器将以多种格式520 - 526之一输出代码，所述格式520 - 526表示被读取到该系统里的原始的源代码。在整个这个过程中，以持久性的存储器528的某种形式来存储完成该过程所必需的数据。

因此，编译过程涉及将指令从一级或者一个表示转换为另一级或另一表示。在一个实施例中，将IR从高级的操作翻译为机器代码的过程能够是一系列优化和代码生成路径，其简单地以越来越多的信息来标志该IR，同时永不改变基本形式。例如，图6 (a) - (d)贯穿了从在源中读取、到高级的与机器无关的IR、到低级的与机器有关的IR的IR的转换。正如这些图所示的，以类似寄存器和实际的机器操作码的目标机细节扩展和标志了IR。然而，形式基本上保持相同。

IR还可以被调节以供不同的SDA配置使用。例如，即使 (JIT) 编译器配置是受到速度和存储器约束的，因此可以通过添加或者除去倾向于降低编译速度或者增加存储器占用面积的属性，来配置在使用SDA的JIT配置创建的JIT编译器中使用的IR的形式。对比起来，优化本机编译器的高端的整体的程序趋向于非常资源密集的，并且要求高代码质量。因此，可以通过添加或除去趋向于限制编译速度或者降低可由编译器得到的存储器量的属性，来配置在使用SDA的本机编译器配置创建的本机编译器中使用的IR的形式。

在这一实施例中，SDA中使用的IR可以是图形数据结构，该图形数据结构被组织为由操作员表示的数据流操作的线性流、数据流源集合和数据流目的地集合的。数据流信息和副作用可以被明确地包括在数据流集合中。可以使用源级类型信息对IR进行强归类，以允许健壮代码生成和类型检查。IR还可以包含充分描述该程序所必需的数据流和控制流链接中的某些或者全部。换言

之，如果提供了所有必需的链接，而不再需要其他数据结构，例如流程图，或者异常处理树/图。

另外，由操作使用的或者定义的任何资源，无论显式的还是隐式的，均可以在操作源或者目的地表上出现。数据流分析和编译器结构一般都可以被简化，因为需要被分析以确定操作的全部的副作用的仅仅是操作码。因此，容易添加新的操作，因为它们是通过简单地以新的方式将操作码的相同的小集合重新组合来实现的。

IR可以包括单一模型，用于经由与操作码相关联的单一“标记符”对存储器进行二义性消除，并确定别名分析。标记符可以或者是保守的，或者使得在大范围的分析期间均非常精确。编译器的所有阶段可以简单地认可该标记符，并且为该操作的副作用要求“Oracle”。

以下部分描述了适于在SDA的所述实施例中使用的IR的一个实施例的实施细节。

#### 示例性的IR

用于SDA实施例的示例性的IR可以是作为归类的、线性的字节组流来实现。每一字节组可以具有一组输入操作码（源表），操作符，以及一组输出操作码（目的地表）。副作用（例如，所有副作用）对于包括实际的或者隐含的寄存器用途/定义和符号用途/定义信息的指令能够是显式的。间接寻址将具有附加给它们的干涉信息。

在有或者没有流程图的情况下，IR均可以存在；然而，全局优化可能要求一个流程图。异常处理区域图表和优化区域图表还可以被用于补充IR信息。IR应具有足够的信息来构建流程图、循环图和至异常处理区域图表的恢复映射。由于流程图可能通常都是存在的（以及循环，以及再小程度是异常处理区域图），所以设计能够就像这些图属于核心IR的一部分的系统中一样，提供易于访问这些图的途径。

IR可以被设计为直接支持基于树的和基于SSA的优化。这可以通过在叶字节组上添加def字段来实现。在其最简单的形式中，可以将表达式temps链接在一起，以形成表达式树。这可以是被预降级的代码的不变量。这一表达式线程可以被不需要使用表达式视图的分析所忽略。转换可以将它们保持为正确的状态，并且可以使用表达式优化器进行净化。充分发展的SSA可以使用相

同def字段并且包括类似PHI指令的专用操作。在SSA情况中，表达式temps可以简单地变为整体SSA图的一个子集。

为了支持inline\_code（纳入代码），编译器的早期阶段可以看到概括\_asm序列的数据流效果的“概括”指令。实际的\_asm指令序列可以作为副表挂起，

5 直到非常迟的降级阶段纳入该序列。

尽管IR的各范例示出降级过程，但是IR也可以被用于升级过程。例如，一个工具可以使用二进制码作为输入，并且为该二进制码结构IR。

#### 操作码

操作码可以是该指令的叶节点，并且在该指令的源和目的地表上出现。可以使得所有副作用在该指令上是显式的，以致提供操作码来描述某些或者所有实际的或者潜在的资源使用，包括立即数，寄存器，存储器和状态码。每一叶节点可以具有与它相关联的、表示它的抽象类型的类型，其随后被映射到关于被降级的指令的机器类型。

#### 寄存器

15 寄存器操作码既能够指定实际的物理寄存器，也能够指定虚拟寄存器。

#### 临时寄存器

临时操作码能够使用两种形式之一：1) 表达式临时寄存(即，临时寄存器)可以是单一的定义，单一的用途，在表达式内使用的临时寄存；或者2) 通用临时寄存(即，临时变量)能够被用于questionOps、CSE、标量替换和归纳变量等等的编译器引入，并且能够具有多个定义和用途。

表达式临时寄存可能仅仅存在于寄存器中，并且可能永不跨越基本块边界存在。通用临时寄存能够存在于寄存器或者存储器中，并且能够跨越基本块边界存在。可能两种情况下都不获取它们的地址。

#### 物理寄存器

25 物理寄存器是一种实际的机器寄存器。这包含整数、浮点、多媒体、矢量以及专用寄存器。

#### 命名寄存器

命名寄存器可以是一种用户变量，该用户变量作为用于或者已经被分配给物理的机器寄存器的候选者。

30 存储器

存储器操作码可以指定存储在存储器或者抽象存储器中的值。 抽象存储器能被用于显式的干涉计算。

命名存储器

命名存储器可以指定存储在存储器中的用户变量。

5 矢量存储器

矢量存储器可以在用于矢量运算的向量操作数中使用。

间接寻址存储器

存储器间接寻址能够表示任何目标机上的最强大的编址方式。 间接寻址还可能包含干涉信息。

10 抽象存储器

抽象存储器可被用于注释具有显式存储器用途/定义副作用、而不是通过直接引用的指令（即，干涉信息）

地址

地址操作码可以指定代码或者数据的位置。

15 有效地址

有效存储器地址操作码能够表示任何目标机上的最强大的编址方式。

数据地址

数据地址可以是被用于存取数据的有效地址的简单形式。

代码地址

20 代码地址可以是被用于非本地代码的有效地址的简单形式。

标记

标记可以是被用于本地代码的有效地址的简单形式。

状态码

IR可以使用状态码的抽象概念。

25 寄存器组

寄存器组能被用于表示删除集合（kill sets）和副作用。

立即数

立即操作数可以指定可以在指令中出现的已知值。 该值可以始终被编译器知道，或者仅仅在后来的阶段中被确定，包括降级，链接和加载状态。

30 整数立即数

整数立即数能够被标准化为最大的整数大小。

浮动立即数

浮动立即数能够被标准化为内部/可移动的形式。

符号立即数

5 符号立即可以是表示由编译器计算的或者可能由链接程序/运行时决定的恒定值的符号。

运算 (与机器无关的操作码)

运算可以被分为几个类别。 这些类别可以被组合或者被分解为截然不同的、可变尺寸的、诸如下列的各种指令:

10     \_ 算术运算

      \_ 具有溢出运算的算术运算

      \_ 逻辑运算

      \_ 指针运算

      \_ 结构运算

15     \_ 对象运算

      o 分配

      o 调用 (虚拟, 实例)

      o 字段访问/更新

      \_ 数组运算

20     o 分配

      o 读/写

      o 没有边界校验的读/写

      o 长度运算

      \_ 矢量运算

25     \_ 分支运算

      \_ 调用运算

      \_ 固有调用运算

      \_ 返回运算

      \_ 切换运算

30     \_ 异常运算

- 5
    - \_ 类型转换(动态/静态计算)
    - \_ 类型测试运算
    - \_ 安全性运算
    - \_ 线程运算
    - \_ 同步运算
    - \_ 校验运算
      - o 空指针校验运算
      - o 边界校验运算
      - o 类型测验运算
  - 10
    - \_ 概括伪操作
    - \_ 数据运算
  - 15      块  
当提供该流程图的时候，特定的块伪指令可以表示基本块的开始和结束。  
标签  
标签可以是由用户定义的或者由编译器产生的。  
附注  
附注可以是由用户定义的或者由编译器产生的。  
注释  
注释可以是由用户定义的或者由编译器产生的。通常，它们包含本机编译器和JIT编译器可以用于改善代码质量的附加的语义信息。  
20      列表  
列表IR节点可以是被用于创建由控制指令指定的流程中的链接的辅助节点。  
分支表  
分支表节点可以被附于标签上，并且表示所有到达分支。  
事例列表  
事例列表节点可以被附于切换运算上，并且表示所有事例值和分支。  
25      调试信息（行和列）  
每一指令可以包含描述源行和列信息的调试信息。当对代码扩展 / 降级或者执行优化的时候，可以为了调试的目的提供该信息。  
30      类型系统

IR还可以包括用于对为IR的各种形式中的一致性进行检验的类型表示。具体来讲，可以提供归类的中间语言，其在表示以包含归类和未归类的语言、松和强归类的语言以及有和没有无用单元收集的语言在内的多种源语言所编写的程序中使用。另外，可以在SDA中提供类型检验器体系结构，其允许根据用于程序组件和/或编译阶段的源语言，使用不同的类型和类型校验规则。

例如，可取的是，将一个高级优化器应用于以多种语言编写的程序。这些语言可以具有不同的基元类型和基元运算。一种语言例如可以包含用于复数算术的类型和运算，而另一语言可以包含专用于计算机图形学的类型和运算。通过允许中间表示被不同类型系统参数化，优化器可被用于具有不同基元类型和运算的语言。

另一范例可以包含一个程序，在该程序中，某些组件是以一种语言的强类型子集编写的，而其他组件是以不是类型安全的完整语言编写的。可取的是，为第一组件集合进行更多的误码检验。这可以伴随着为不同的组件使用不同类型的校验规则。

还有另一个范例是在编译期间丢弃类型信息。类型检验器和编译器可以允许在后期丢弃类型信息，同时强制在前期维护精确信息。这可以伴随着与用于不同编译阶段的不同类型校验规则相结合地使用未知类型。

在SDA的一个实施例中，可以在类型类分级结构中定义多个类型表示，以致各种语言的类型系统可以由一个归类的IR表示。可以将抽象基本类定义为用于所有类型的‘Phx::Type’。基本类例如可以包含用于各种类型的、例如实际的、符号的或者未知的（或者可变的）类型的大小信息‘sizekind’。为了指定类型分类，基本类还可以包含‘typekind’。另外，可以将外部类型作为抽象类型提供，该抽象类型将外部定义封包，以便提供从类型IR到原始源代码的倒映射。

在基本类之下，被定义为‘Phx::PtrType’的类可以表示指针类型。也可以定义各种指针。例如，被管理的无用信息收集指针（指向无用信息收集对象中的一个位置的指针），被管理的非无用信息收集指针（指向非无用信息收集对象中的一个位置的指针），未被管理的指针（例如，往往在以C++编写的代码中发现的指针），参考指针（指向无用信息收集对象的基地址的指针），以及空值。

在分级结构中的同一级中，被定义为‘Phx::ContainerType’的类可以表示容器类型，例如包含内部成员的类型。内部成员可以具有字段、方法及其他类型。被定义为‘Phx::FuncType’的类可以表示函数类型，包含任何必需的调用约定，自变量列表以及返回值类型列表。此外，被定义为‘Phx::UnmgdArrayType’的类可以表示未被管理的数组类型。基于分级结构中的‘Phx::ContainerType’，可以定义四个以上的类。被定义为‘Phx::ClassType’的类可以表示类类型，被定义为‘Phx::StructType’的类可以表示struct（结构）类型，被定义为‘Phx::InterfaceType’的类可以表示接口类型，并且被定义为‘Phx::EnumType’的类可以表示枚举类型。基于分级结构中的‘Phx::ClassType’，被定义为‘Phx::MgdArrayType’的附加类可以表示被管理的数组类型。

类‘primtype’可以被定义为struct类型的一个特定实例。‘primtype’可以包含诸如int(整数)、float(浮点)、unknown(未知)、void(无效)、状态码、unsigned int(无符号整数)、xint等等之类的各种类型。这些表示可被用于类型IR中的HIR或者LIR两者中。

另外，目标专用基本类型可以被归入类型表示中。某些语言具有复数算术类型，如果让类型系统知道复数算术类型，则可以高效地处理它们。以“MMX”指令作为例子。这样一种指令是被嵌入某些版本的x86处理器内部的用于支持对于多媒体和通信数据类型的单指令/多数据运算的附加指令集。该类型系统可以被定制，以便以最小的类型表示改变来识别和使用这些指令。

上述类型的类型表示的实施例还可以包含“未知(unkown)”类型，其可以表示任何类型，并且可选的是具有与它相关联的大小。大小可以是该值的机器表示的大小。未知类型允许编译器通过改变从一种特定类型到一种未知类型的类型信息，以受控的方式丢弃类型信息。它允许编译器生成取决于被操作的值的大小的代码，而即使是在该类型是未知的时候。其他类型可以使用未知类型，以致未知类型还允许不完全类型信息（其中某些但不是所有信息是已知的）的表示。

例如，假定一个指向int(整数)类型的指针。在降级的某一阶段，可能可取的是丢弃类型信息int(整数)。未知类型允许编译器将整数类型替换为未知类型替换。然后类型检验器无需校验所感兴趣的指针是指向一个正确的类型。实质上，它冒着这样的风险：将以在运行时不至于不利地影响程序功能的方式

处理所指向的值。

使用未知类型的另一范例是用于定义用于函数的类型。如果一个具有指向未知的类型指针的自变量的函数被调用，则如果该自变量先前具有指向整数的类型指针，则编译器必须信任正在传送正确的类型。废除指针的结果可能被  
5 知道或者可能不被知道是一个整数；然而，它可被作为整数使用。更复杂的范例是在从虚函数调用的高级的中间表示到低级的中间表示转换的期间引入中间临时变量。虚表(vtable)被广泛地用于实现面向对象的语言中的虚调用。在低级的中间表示中作出一个虚函数调用的第一个步骤是取出存储器对象中的第一字段。第一字段包含一个指向vtable的指针。然后，取出的结果被分配给一个临时变量。结构临时变量的类型(表示一个指向vtable的指针的类型，其中该vtable可能具有许多字段)可能是很复杂并且很难于表示的。代之以，编译器可以简单地给中间临时变量分配“指向未知的指针”。因此，未知类型的使用简化了编译的后期阶段，在所述后期阶段中，保持详细的类型信息是不必要的，或者可能对于编译器实现者来讲成为显著的负担。

15 图7图示出用于对编译的各阶段的IR进行类型校验、并从而在降级的各级对类型IR进行类型校验的编译器系统的一个实施例。源代码700表示各种源语言中的任何一种。源代码700被翻译为归类IR的HIR702。在这种情况下，源语言的类型表示被翻译为归类IR内部的类型表示。

20 HIR在整个编译过程期间被降级。为了这一例证说明的目的，示出了高(HIR)702、中(MIR)704和低(LIR)706级表示。然而，该实施例不是受到如此限制的。可以对任何数目的编译阶段进行类型校验。

25 每一表示级的IR可以被类型检验器708类型校验。类型检验器708实现用于向编译过程的每一阶段、并从而向IR的每一表示施用一个或多个规则集合710的算法或者过程。基于各种属性、例如源语言、编译阶段、什么归类强度等等来选择规则集合710。

30 例如，假定源代码700包含以C++程序设计语言创建的代码。C++源代码700首先被翻译为归类IR的HIR702。如果期望的话，在该时刻，类型检验器708可以与HIR 702交互，以便确定任何数目的属性。这样的属性可以包含编译阶段(HIR)，提供的源代码类型(C++)，该语言是否被归类(是)，它是松归类还是强归类(松)，等等。基于这些属性，类型检验器可以选择一个适当

的规则集合。一旦选择一个规则集合，类型检验器依据该规则集合类型校验HIR。一旦HIR被降级为MIR或者LIR，可以再次访问所述属性，并且相同或者不同的规则集合可能是适当的。

在一个实施例中，可以为类型检验器提供三个类型校验规则集合。一个  
5 集合可以对应于“强”类型校验，例如往往适用于C#或者CIL类型校验。另一集合可以对应于“弱”类型校验，其可能是比“强”类型校验更宽松的类型校验。例如，弱类型校验规则集合可以允许类型指定（cast）。类型指定是当为了单次使用而使一个类型的变量像另一类型的变量似地起作用。例如，int（整数）类型的变量可被强迫作为char（字符）。下列代码使用类型指定来打印字母  
10 “P”。

```
int a;  
a = 80;  
cout << (char) a;
```

因此，即使“a”被定义为类型int并被赋值80，但由于类型指定，cout语  
15 句将变量“a”作为类型char来处理，因此显示‘P’（ASCII值80）而不是80。

最后，一个集合可以对应于“表示”校验。“表示”校验可以允许例如通过使用未知类型在部分中间程序表示中丢弃类型信息，并且可以包含指定这样的类型信息可以在什么时候被丢弃或者未知类型可以在什么时候被另一类型代替的规则。例如，返回类型值Void（无效）的函数结果可以被禁止分配给未  
20 知类型的变量。

另外，能够在编译的单一阶段使用超过一个规则集合。例如，假定源代码  
700包含单一语言，但是包含被强归类的部分和被松归类的某些部分。类型检验器可以在某些强类型部分为HIR使用一个规则集合，并且为宽松类型的代码部分使用另一规则集合。

图8是一种供此处所述的IR使用的类型检验器的方框图。类型检验器800  
25 可以被视作接收与不同的源语言和/或不同的编译阶段相对应的任何数目的规则集合。在图8中，为类型检验器800提供了四个规则集合802 - 808。规则集合802表示用于具有强归类语言的HIR的规则集合，规则集合804表示用于具有弱归类语言的HIR的规则集合，规则集合806表示用于无归类语言的HIR的规则集合，  
30 并且规则集合808表示用于LIR的规则集合。程序模块810表示在HIR中强归类

语言，并且程序模块812表示在被降低为LIR之后的程序模块810。

类型检验器800基于被类型校验的程序模块的属性选择适当规则集合，并且将选择的规则集合应用于使用组合的过程或算法的程序模块。例如，类型检验器800可以选择规则集合802（表示用于强归类语言的HIR的规则集合），

- 5 以便类型检验程序模块810（表示在HIR中强归类的语言）。其后，类型检验器800可以选择规则集合808（表示用于LIR的规则集合），以便类型检验程序模块812（表示在LIR中强归类的语言）。

所述的类型检验系统的规则集合可以被容易地扩展至全新的语言，并且也可以容易地扩展至现有语言的新特征。例如，如果引入了一种新的语言，则

- 10 只不过是为该新的语言创建一个新的规则集合。由于规则集合是与类型检验器或者编译器系统自身相独立的，并且被设计为将规则集合视为独立的实体，因此可以在无需重新分配或者更新现有类型检验系统或者编译器的情况下为新的语言分配新的规则集合。同样地，如果一个新的特征被添加给一种现有语言，例如将XML支持添加给C++，则在编译的各阶段与C++相对应的规则集合
- 15 可以被容易地动态重构为处理新的特征。此外，不需要更新或者分配新的核心系统。

规则集合还可以允许对于类型的约束条件。例如，是否允许在从另一个类继承下来一个类的时候对一特定的类型进行子归类，可以是一个在规则中所描述的约束条件。另一约束条件可以是总约束条件，例如可能希望指示数据可以被转换为包含该数据的虚表。其它可以包含大小约束条件，或者是指示需要完全相同的基元类型的基元类型约束条件。类似于规则集合的任何其他部分，可以依照要求添加新的约束条件。

- 由类型检验器使用的规则集合可以经由与用于创建规则集合的应用程序的编程接口来结构。该应用程序可以结构规则，以致以基元类型与被分配给归类IR的单独指令的规则的分级结构形式来表示规则集合。可以以类型图的形式提供该分级结构，类型图明确地表示与特定程序模块或者编译单元相关的类型的各元素。诸如符号和运算之类的IR元素将与类型系统的元素相关联。类型图节点将描述基元和被结构的类型以及它们的关系，诸如组件，嵌套类型，函数签名，接口类型，分级结构的元素，及诸如源名称和对于模块/组件外部类型
- 25 元素的参考之类的其他信息。

## 异常处理

如此处所述的供SDA使用的IR可以支持各种语言专用异常处理模型，诸如C++，Microsoft CLR，以及Microsoft Windows结构化异常处理（SEH）。

图9图示出用于通过编译器后端940在IR 930中为多个源语言（905 - 908）实现统一的异常处理的系统900。如图9中所示，系统900为多个源代码表示905 - 908中的每一个包含一个中间语言(IL)表示910 - 913，源代码表示905 - 908是被将多个IL表示910 - 913翻译为单一IR 930的IL阅读程序920分析或者读取的。所述IL表示是比IR 930更高级的中间表示，并且可以以任何数目的公知的中间语言来表示，诸如CIL (Microsoft CLR)（用于C#，Visual Basic，JScript，C，和FORTRAN）和C+ + IL（用于C++）。即使用于为多个语言生成统一的异常处理框架的系统900被显示为具有用于多个源语言的单一阅读程序过程，但是可以实现多个这样的阅读程序，每一个阅读程序对应于IL表示910 - 913中的一个或多个。

图10A图示出一种通用的整体方法1000，该方法用于使用IL阅读程序920为以多个不同的源语言表示的异常处理结构产生一组统一的中间表示。在1010，软件的中间语言表示（例如，源码文件的中间语言表示）是由阅读程序920接收到的，并且在1015，该文件被读出，或者被分析，以识别IL代码流内部的异常处理结构（1020）。然后在1030，阅读程序920（其还可以被认为是虚拟机）生成先前在1020识别出的异常处理结构的单一的统一IR。这样一种异常处理框架能因此被用于简化编译器后端的处理，诸如编码优化和代码生成。

具有异常处理结构的软件的统一IR可以显式地表示软件的异常处理控制。图10B示出用于根据软件的统一IR生成可执行的方法1050。这种方法例如能被编译器或者其他软件开发工具在为软件生成可执行的版本（例如，机器专用代码或者其他目标代码）的时候使用。

在1060，统一IR被读取（例如，通过编译器或者其他软件开发工具）。例如，能使用通过图10A中的方法生成的统一IR。可以依照要求执行对于统一的中间表示的其他转换，翻译，或者优化。

在1070，软件的计算机可执行版本被生成（例如，通过编译器或者其他软件开发工具）。软件的计算机可执行版本基于统一IR实现软件的异常处理控

制流。

图11图示出用于在以多IL表示的形式表达的多个源语言内部产生异常处理结构的简单的和统一的IR的系统的另一实施例。如图11中所示，在Microsoft的.NET框架内支持的源语言组1110（例如C#，C，Microsoft Visual Basic，Jscript，  
5 和FORTRAN）首先被翻译为CIL表示1140。然而，由于它与其他源语言的差异，C++是以通常被称为CIL的另一种中间语言 1130来表示的。C++ IL和CIL内部的控制流和异常处理模型是以根本上不同的方式来表示的，因此可能需要为C++ IL和CIL表示提供独立的IL阅读程序（1135和1145）。

10 阅读程序1135和1145两者都可以使用在它们的相应的阅读程序内部实现的适当的算法来分析或者读取它们的相应的中间语言代码流，以便使用将在后端  
1160提供的统一的异常处理指令框架1150来表示中间语言代码流内部的异常处理结构或者指令或者表达式。

### 代码生成

15 为了使用SDA为任何数目的执行体系结构产生软件开发工具，SDA可以包含能够根据公用IR产生任何数目的机器相关表示的代码生成程序。图12中图示出用于这样的代码生成的系统的实施例。在该范例中，为重定目标工具  
1202提供了具有目标专用数据的目标执行体系结构规范1200。替换地，可以为重定目标工具1202提供规范1200和具有目标不可知数据的第二规范1204两  
者。

20 重定目标工具1202是对用于与一个或多个代码生成组件有关的数据的规范1200/1204进行分析的软件。基于规范1200/1204中的数据，重定目标工具创建一个或多个组件。每一组件提供用于创建代码生成组件的数据。

例如，在图12的实施例中，产生了令牌化语法1206（例如，以lex格式），  
25 和语法分析程序语法1208（例如，以yacc格式），以及C源和主文件系列1210。

在使用lex的一个范例中，为lex编译器1212提供了令牌化语法1206，用于创建对产生编译器的词法分析组件来说是必需的源代码和主文件1214。Lex是专门设计用于创建编译器词法分析组件的语言设计。

在使用yacc的一个范例中，为yacc编译器1216提供了语法分析程序语法  
30 1208，用于创建对产生编译器的语法分析组件来说是必需的源代码和主文件1214。 yacc提供了用于指定源代码输入的结构的普通工具，同时还提供了随着

每一个这样的结构被识别出来而被调用的代码。

最终负责产生对于构建软件开发工具来讲所必需的组件的源代码是通过将重定目标工具产生的C源和主文件系列1210/1214与SDA配置内部包含的公用框架1218集成而产生的。例如，词汇和语法分析组件是通过对C源和主文件系列5 1214与公用框架源代码和主文件1218的集成进行编译而产生的。同样地，其他软件开发工具组件可以通过对C源和主文件系列1210与公用框架源代码和主文件1218的集成进行编译来产生。

因此，为编译器1222提供表示C源和主文件1210/1214与公用框架源代码和主文件1218的集成的源代码1220，以产生编译器1224，该编译器1224的目标是在符合SDA内部的公用框架的规范1200中所述的目标执行体系结构。  
10

如上所述，软件开发工具组件包含合法化表。这样的表包含足以识别目标指令的具体形式的信息。合法化框架可以提供形式下标，该形式下标是指令选择、指令合法化、编码和列表组件使用的。

图13是一种以软件实现的、用于适于与此处所述的技术一起使用的代码生成的方法的实施例的流程图。目标执行体系结构规范在1302处被处理，以便创建由源代码表示的多个组件。然后，在1304处，源组件被集成到由源代码表示的框架中。该框架可以具有或包含有SDA核心。如果期望的话，也可以在块1306处纳入自定义码。然后集成的源代码在块1308处被编译，以便通过代码生成组件创建软件开发工具。  
15

## 20 核心数据结构扩展

在SDA的另一实施例中，为扩展性地配置SDA核心数据结构提供了一种方法和系统，所述SDA核心数据结构的扩展字段可以取决于正在被结构该SDA的该配置的软件开发场景，或者是取决于目标软件开发工具的个别期望特性的任何其他因素。图14A描述了一种用于扩展核心类定义以便通过扩展核心框架构建工具的全过程。首先，在1402处遇到指示扩展的数据，并且在1404处，  
25 按照扩展所作出的指示，扩展软件开发工具的类。

图14B描述了用于通过使用与软件场景有关的扩展来扩展SDA核心框架以便构建软件开发工具的全过程。在1410处，一种简化的对象定义语言(ODL)可被用于定义核心类。然后，在1420处，可以基于任何数目的因素、诸如软件开发场景，来确定用于特定SDA的配置。软件开发场景可以指示待创建的软  
30 件

件开发工具的类型，目标执行体系结构，目标输入语言等等。然后，基于被考虑的某些或者所有因素，在1430处，所述对象描述语言可被用于定义扩展，以便表示扩展核心类所需要的附加的或者不同的类成员。

在1440处，扩展可以是与核心类相关联的，以便适当地扩展核心类定义。用于对象描述语言的语法应指定将核心类定义为可扩展的或不可扩展的，并且随着扩展被选择的核心类，进一步关联特定的扩展类成员的集合。此外，预处理器翻译程序可被用于将数据或者对象描述语言翻译为程序设计语言。

在这样的预处理之后，在1450处，扩展类定义可以通过扩展核心框架，被进一步编译，并被用于实现特定配置的其他软件开发工具。

通过使用上述过程，可以单独地提供多个不同的扩展定义，并且每一扩展可以根据需要简单地扩展核心或者基本类，而无需维持任何复杂的继承关系。提供核心类的特定扩展的程序员无需知道该核心类的其他扩展。这不仅简化了定义扩展的工作，并且，扩展核心类的用户仅仅需要了解核心类名来使用扩展核心类。因此，程序员可以免于在使用扩展类定义的时候记忆类定义之间的复杂的分级关系的辛劳。

扩展核心框架程序的一个方法可以是能够理解核心程序的源码文件，并能够按照需要，通过使用对象描述语言定义扩展来静态地扩展核心类，所述扩展随后被处理，以便自动地将该扩展引入核心类来产生扩展类。替换地，扩展也可以通过手动地将该扩展直接添加至源代码语言中的源代码来生成。图15A图示出这一方法，借由该方法，扩展1520、1530和1540被添加到核心框架文件1510中，以便扩展它，并且随后该扩展1520、1530和1540被作为现在被扩展的核心框架文件1510的一部分进行编译。

然而，这一方法可能不适合于所有的用途，因为提供诸如1520、1530和1540之类的扩展的定义的程序员将需要能够访问核心框架1510的源代码。这在核心框架1510的供应商希望对核心框架源代码保密的情况下并不可取。在该情况下，可以使用图15B中所描述的第二种方法，借由该方法，核心编译器框架1550是作为与扩展1560、1570和1580相独立的文件来编译的。

在第二种方法中，扩展1560、1570和1580与核心框架1550可以被改为具有与彼此的链接，以便在运行时，扩展被链接到核心框架，来适当地扩展核心框架。链接可以作为简单的链接列表来实现，该链接列表规定哪些扩展将被用

于扩展特定的核心类。这还可以通过使用简单的命名规则来实现，该命名规则按照需要、并在需要时适当地将扩展与核心类联系起来。与第一种方法相比，第二种方法在运行时可能要求与链接方面有关系的额外的系统开销处理，因此可能是一种更慢的实现方式。而另一方面，第二种方式却提供了这样的  
5 灵活性：允许由不能访问该核心框架的源代码的开发人员扩展核心类。

图16图示出一种用于如上参照图15A所示的、在编译时间之前静态地扩展与核心框架程序相关的类的方法。可以使用对象描述语言定义核心类和它们的扩展。核心类和扩展的定义无需同时地或者一起生成。然而，添加扩展往往要求对核心程序的源代码进行某些访问。  
10

一旦获得这样的类定义，则在1610处，核心类和它们的扩展的定义将被ODL预处理器一起处理，该ODL预处理器能够将对象描述语言表示翻译为源代码表示。因此在1620处，由ODL处理器执行的预处理的结果往往是主文件，并且可能是表示在诸如C++这类的源代码语言中的核心类和它们的扩展的定义的其它代码。进一步来讲，在1630处，具有包括核心类成员和扩展类成员在  
15 内的扩展的类定义随后和与现在被扩展的核心框架有关的代码一起被编译，以生成自定义的配置的软件开发工具。

图17图示出用于实现图16中的过程的示例性系统。如图17中所示，对于核心类定义1720的多个扩展定义1710可以作为对象描述语言文件来存储。可以提供ODL预处理器1730，其能够接收分别对应于核心类定义和扩展定义的文件1710和1720。该预处理器还应能够将文件1710和1720从它们的对象描述语言形式翻译到源代码表示1740。源代码表示可以采用能够最终被编译为可由计算机处理器执行的形式的任何语言的形式。由预处理器1730生成的源代码  
20 1740可以包含在其中通常存储有类定义的主文件。可以提供与由预处理器发出的源代码1740的语言相适应的源代码编译器1750，用于编译源代码表示  
25 1740，以创建诸如编译器及其他软件开发工具之类的核心软件程序1760的定制的扩展版本。

图18图示出用于通过在运行时将扩展链接至适当的核心类来扩展可扩展核心框架软件程序的核心类定义的方法。可以使用对象描述语言单独地表示核心类定义和扩展。该描述语言可以适用于表示核心类定义是动态地可扩展的。  
30 同时，这样的一种语言可以适用于表示特定核心类定义和它们的扩展之间的关

联。下文中将更加详细地说明用于一个这样的合适的语言的语法。一旦定义被表示，则可以在1810处使用ODL预处理器将对象描述语言表示中的定义翻译为1820处的源代码表示。然而，不同于静态过程（图16），在图18中的动态过程中，核心类定义不是由ODL预处理器与它们的扩展的定义一起处理的。  
5 代之以，对应于核心类定义的源代码主文件和对应于类扩展定义的源代码主文件被单独地生成。这些可以是由不同的ODL预处理器生成的，但是并不是必需这样做。此外，在1830处，包含核心类定义的主文件和包含扩展定义的主文件被单独地编译，以创建可由计算机执行的单独文件。然而，在1840处，在运行时期间，所述类扩展定义可以被链接到适当的核心类定义，以便按照定义  
10 来扩展该核心类。

图19图示出用于实现图18中的过程的示例性系统。如图19中所示，所述类扩展定义是以对象描述语言的形式来提供的，并被存储在文件1910中。没有必要如图所示的那样，将每一个扩展存储为一个单独的文件。核心类定义也以对象描述语言的形式提供，并被存储在文件1920中。依据图18中所描述的过程，提供了ODL预处理器1925，用于通过将核心类定义从对象描述语言表示翻译为将被存储为主文件1935的源代码语言表示，来处理核心类定义。类似地，还提供了另一个ODL预处理器1930，用于处理类扩展文件1910，以便生成包括扩展的源代码主文件1940。可以提供源代码编译器1945，用于编译类扩展主文件1940，以生成包含类扩展定义的计算机可执行文件1960。类似地，  
15 可以提供源编译器1950，用于编译包含核心类定义的主文件1935，以生成包含核心类定义的计算机可执行文件1955。然后在运行时，随着对应于核心类1955的可执行文件、和对应于扩展类的可执行文件被执行，在核心和扩展类内部提供的链接1970能够使得核心类被适当地扩展。  
20

### 可交付使用的软件

图20是用于可交付此处所述的技术使用的计算机软件的方框图。可交付使用的软件2000包含SDA 2002。SDA 2002能够包含用于实现此处所述的IR、异常处理模型、类型系统、代码生成器或者核心数据结构扩展技术中的一个或多个的源代码或者对象。SDA 2002能够被配置为产生任何数目的软件开发场景中的各种软件开发工具中的任何一种。配置能够包含、但是不局限于：本  
25 机编译器配置2004，JIT编译器配置2006，预JIT编译器配置2008，分析工具配  
30

置2010，缺陷检测配置2012，优化器配置2014，或者CDK配置2016。

可交付使用的软件能够在适当的计算机系统上执行，以便产生目标软件开发工具。为所述计算机系统上的可交付使用的软件 / SDA 2000/2002提供了组件（或多个组件）2018。组件（或多个组件）2018能够包含、但是不局限于  
5 下列项中的一种或多种：与目标执行体系结构2020有关的数据，类扩展声明集合2022，与用于所述目标软件开发工具的输入语言相关的数据2024，和一个或多个类型规则集合2026。

可交付使用的软件能够基于选择的配置和组件（或多个组件）2018产生软件开发工具。例如，图21是示出图20中所描述的可执行软件的具体范例的方框图。  
10

可交付使用的软件2100包含优化器配置中的SDA 2102。组件2104包含与x86体系结构相关的数据，符合该优化器配置的类扩展声明集合，与输入语言C++相关  
15 的数据2110，以及用于类型校验2112的三个规则集合（例如，一个集合对应于强类型校验，一个集合对应于弱类型校验，并且一个集合对应于表示类型校验）。组件（或多个组件）2104被链接到可交付使用的软件/SDA 2100/2102，以创建优化器2114。优化器2114采用以C++语言编写的源代码作为输入，并且其目标是x86体系结构。  
15

图22示出用于可交付此处所述的技术使用的计算机软件的另一实施例。在这一实施例中，可交付使用的软件2200能够是SDA 2210的二进制码或者其他  
20 计算机可执行文件版本。软件开发组件2220能因此被链接（例如，动态地）至可交付使用的软件2200（例如，在运行时，不用对源代码进行访问）。组件和可交付使用的软件的结果组合能够创建软件开发工具2230。  
20

替换地，也能够如图23中所示地修改先有的软件开发工具2300。软件开发工具2300可以是使用诸如此处所述的各方法中的SDA 2310产生的。新的组件2320能因此被创建，并被链接到SDA 2310，以便创建软件开发工具2330的改进版本。  
25

在又一个实施例中，可以使用SDA创建内部兼容软件工具。图24是使用SDA 2410产生的第一软件开发工具的方框图。包括第二软件工具的功能的一个或多个新的组件2420能因此被创建，并被链接到采用二进制码或者其他可执行格式2430的形式的SDA，以形成第二软件开发工具2440。所述第一和第二  
30

软件工具2400/2440是内部可兼容，因为它们共享SDA的特性（诸如公用IR）。

#### 示例性的IR格式的可扩展性

因为为SDA定义的核心类可以被扩展，所以IR格式自身可以被扩展。例如，新类成员可以被添加给核心类并且以IR格式来表示。这样一种配置对于可扩  
5 缩性很有好处。例如，可能可取的是，为JIT编译器使用轻型的数据结构（例如，为了性能的理由）。另一方面，更综合性的数据结构可能适合于整体编程优  
化编译器。

核心类可以经由此处所述的对象描述语言机制来扩展。因此，核心类可以或者在开发时间、或者在适当的软件组件或者工具的运行时（例如不用访问  
10 核心类的源代码）进行扩展。以这种方式，系统可以支持IR格式的运行时可扩展性。

#### 示例性的软件开发场景独立性

软件开发场景独立性可以以各种方式展现出来。例如，相对于一个或多个软件开发场景的独立性。例如，实现方式可以独立于程序设计语言（例如，  
15 输入语言），软件执行体系结构（例如，处理器或者虚拟机），异常处理模型，被管理的代码场景等等，或者它们的任何组合。

#### 示例性的操作环境

图25图示出用作SDA的实施例的操作环境的计算机系统的范例。计算机系统包含个人计算机2520，该个人计算机2520包含处理部件2521、系统存储器2522、  
20 和将包含系统存储器在内的各系统部件与处理部件2521互连起来的系统总线2527。系统总线可以包括几种总线结构类型中任何一种，仅举几个例子，所述总线结构类型包括存储器总线或者存储控制器，外围总线，和使用诸如PCI、VESA、微通道（MCA）、ISA和EISA之类的总线体系结构的局部总线。系统存储器包含只读存储器(ROM)2524和随机存取存储器(RAM)2525。基本  
25 输入/输出系统2526(BIOS)，包含例如在起动期间帮助在个人计算机2520内部的单元之间传输信息的基本例程被存储在ROM2524中。个人计算机2520进一步包含硬盘驱动器2527，磁盘驱动2528，例如从活动磁盘2529中读取或者向活动磁盘2529中写入，以及光盘驱动器2570，例如用于从CD-ROM盘2571中读取，或者从其他光学介质中读取或者向其他光学介质中写入。硬盘驱动器  
30 2527、磁盘驱动2528和光盘驱动器2570分别通过硬盘驱动器接口2572、磁盘驱

动接口2577和光驱接口2574与系统总线2527连接。 驱动和它们关联的计算机可读介质为个人计算机2520提供数据、数据结构、计算机可执行指令（诸如动态连接库之类的程序代码和可执行文件）等等的非易失性存储。 尽管上面对于计算机可读介质的说明涉及硬盘、可移动的磁盘和CD，但是它还可以包含  
5 可由计算机读取的其他类型的 介质，诸如磁带盒，快擦写存储卡，数字视频盘，Bernoulli盒式磁盘机等等。多个程序模块可以被存储在驱动和RAM 2525中，  
包含操作系统2575、一个或多个应用程序2576、其他程序模块2577和程序数据  
2578。 用户可以经由键盘2540和定点设备、诸如鼠标2542将命令和信息输入  
到个人计算机2520里。其他的输入装置（未示出）可以包含麦克风、操纵杆、  
10 游戏操作台、卫星抛物面天线输入端、扫描仪等等。 这些及其他输入装置往  
往被经由与系统总线耦合的串行端口接口2549连接到处理部件2521，但是也  
可以通过其他接口来连接到处理部件2521，诸如并行端口，游戏端口或者通用串行  
总线（USB）。 监视器2547或者其他类型的显示设备也被经由诸如显示器控制  
器或者视频适配器2548之类的接口连接到系统总线2527。 除了监视器之外，  
15 个人计算机通常包含其他外围输出设备（未示出），诸如扬声器和打印机。

2520可以使用与一个或多个远程计算机、诸如远程计算机2549的逻辑连接，在网络环境中操作。 远程计算机2549可以是服务器，路由器，对等设备或者其他公用网络节点，并且尽管在图25中仅仅图示出一个存储器设备2550，但是远程计算机2549通常包含相对于个人计算机2520描述的许多或者所有这些单元。  
20 图25中描述的逻辑连接包含局域网(LAN) 2551和广域网(WAN) 2552。这样的联网环境在办公室、企业范围的计算机网络、内部网和因特网中是平常的。

当被用于局域网联网环境的时候，个人计算机2520经由网络接口或者适配器2557与本地网络连接。 当被用于广域网联网环境的时候，个人计算机2520通常包含调制解调器2554或者其他用于建立在宽域网2552、诸如因特网之上的通信的其他装置。 可以是内部的或者外部的调制解调器2554经由串行端口接  
25 口2546与系统总线2527连接。 在网络环境中，相对于个人计算机2520描述的程序模块或者是它的一部分可以，在被存储在远程存储器存储设备中期间。 所示出的网络连接仅仅是范例，并且可以使用在计算机之间建立通信链路的其他手段。

### 30 备选场景

尽管已经举例说明和描述了所示例的实施例的原理，但是对于本领域中的技术人员来讲，应该容易地明白的是：能够在不背离这样的原理的情况下在场景和细节方面对本发明作出修改。

考虑到许多可行的实施例，应被意识到的是：被举例说明的实施例仅仅包含范例，而不应该被视为对本发明的范围的限制。更确切的说，本发明仅仅由以下的权利要求书限制。因此，我们主张本发明应为被归入这些权利要求范围之内的所有这样的实施例。

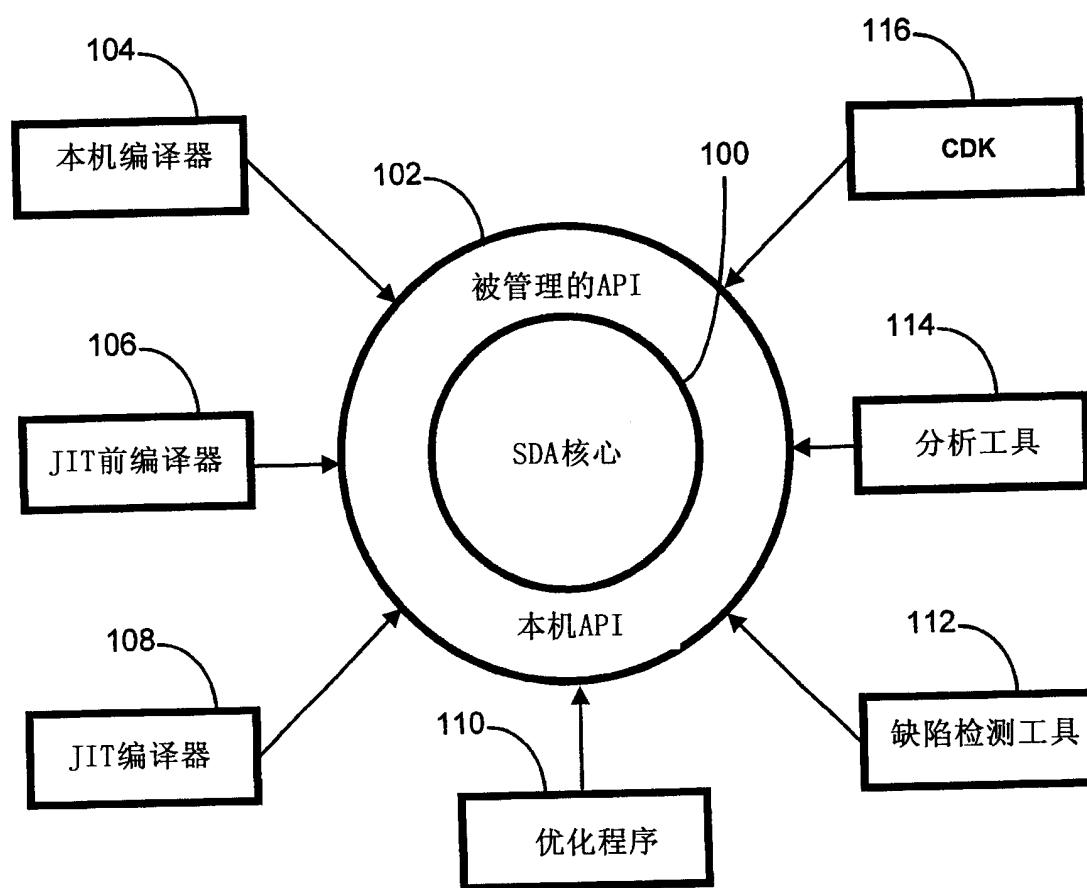


图 1

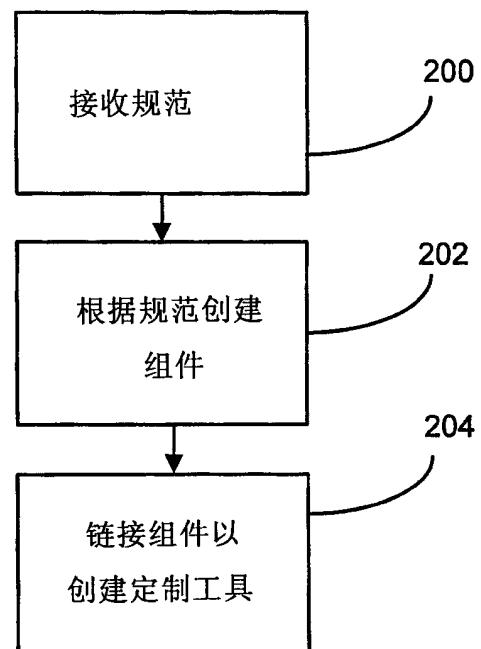


图 2(a)

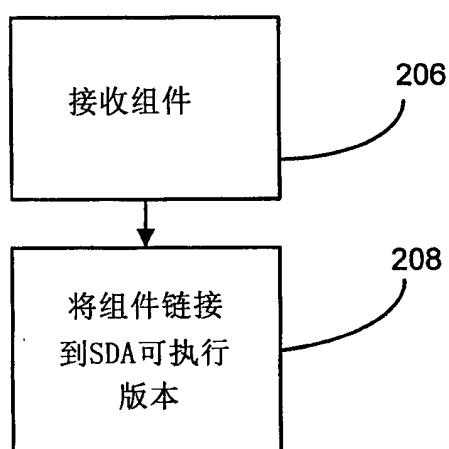
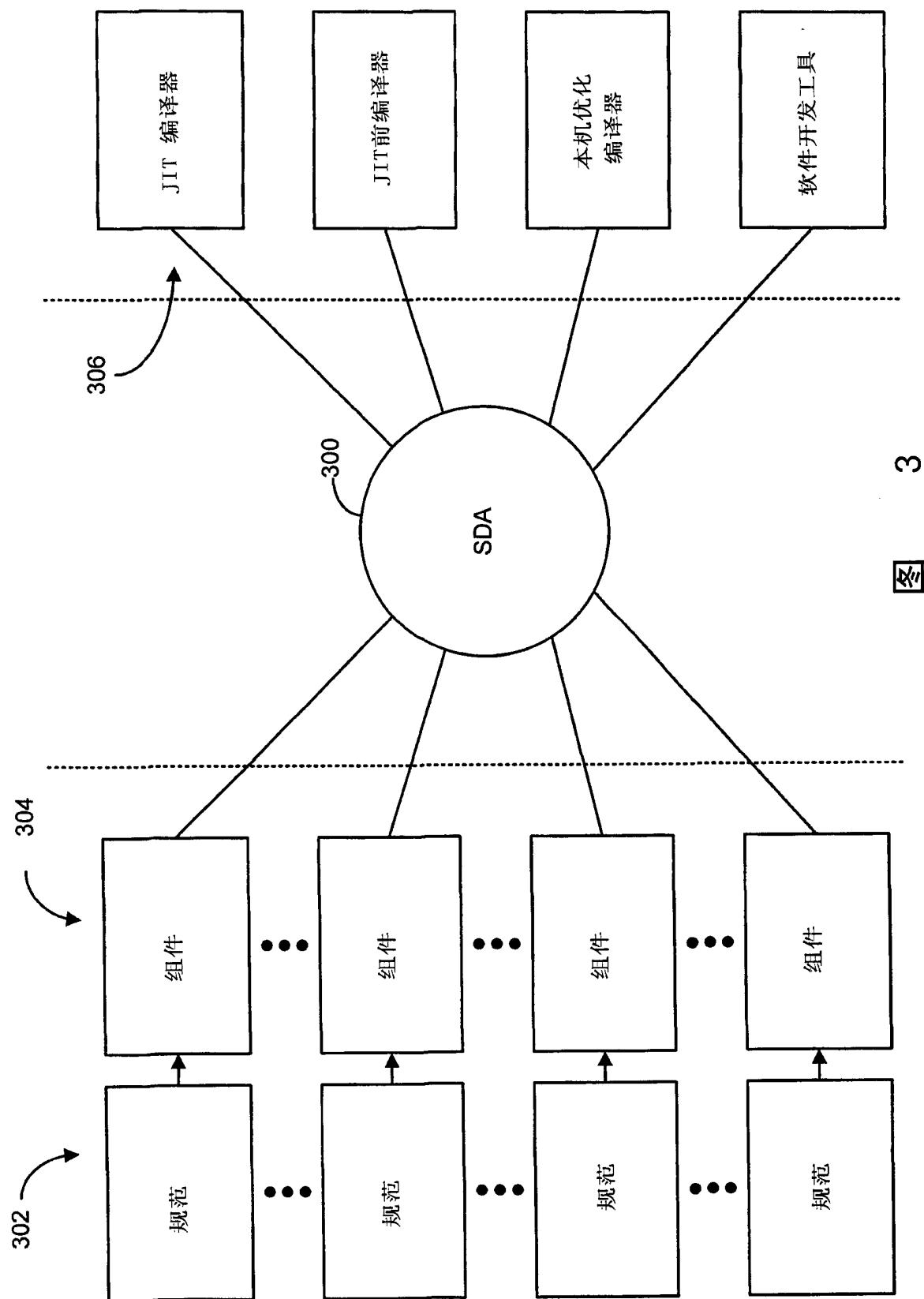
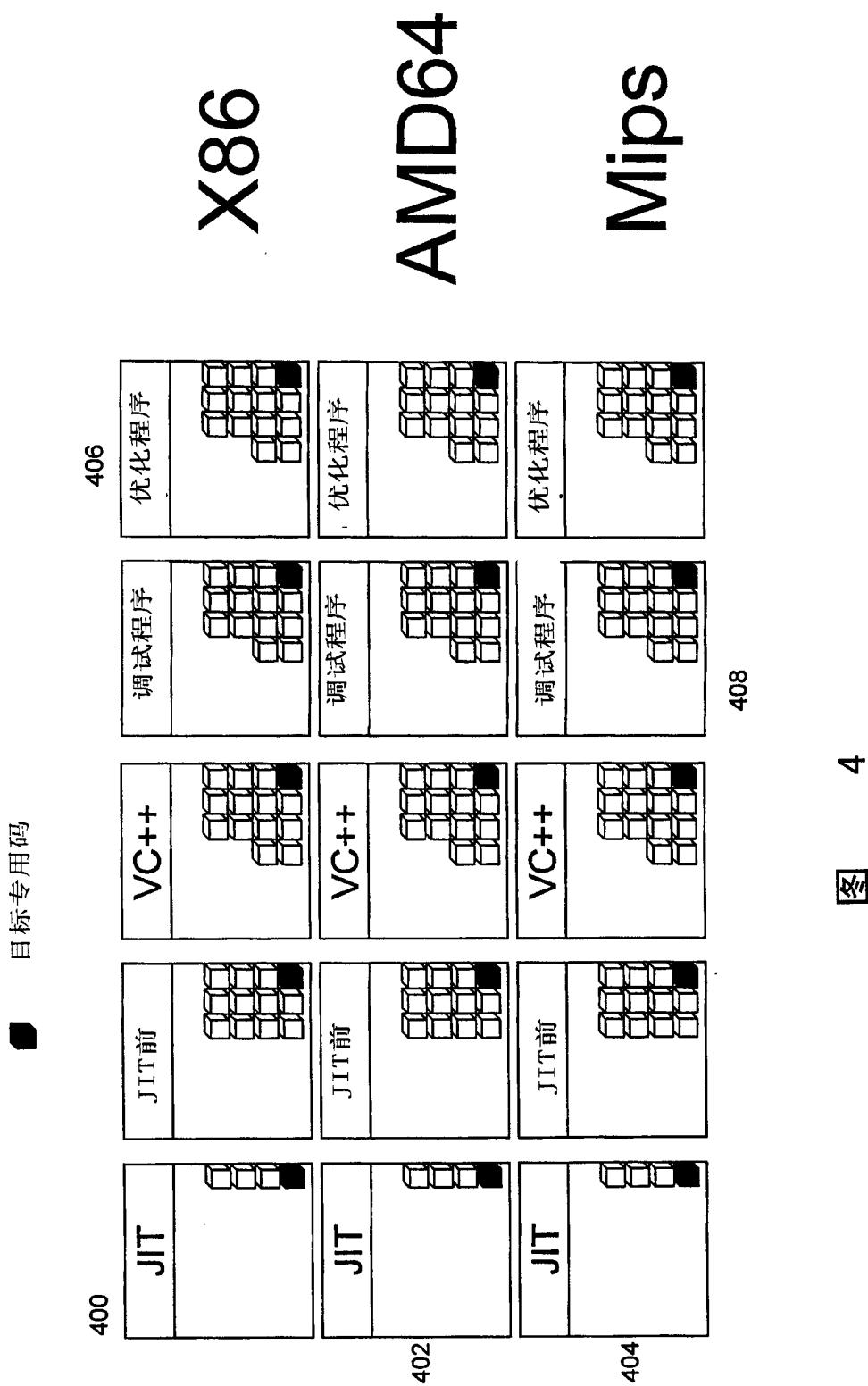


图 2(b)





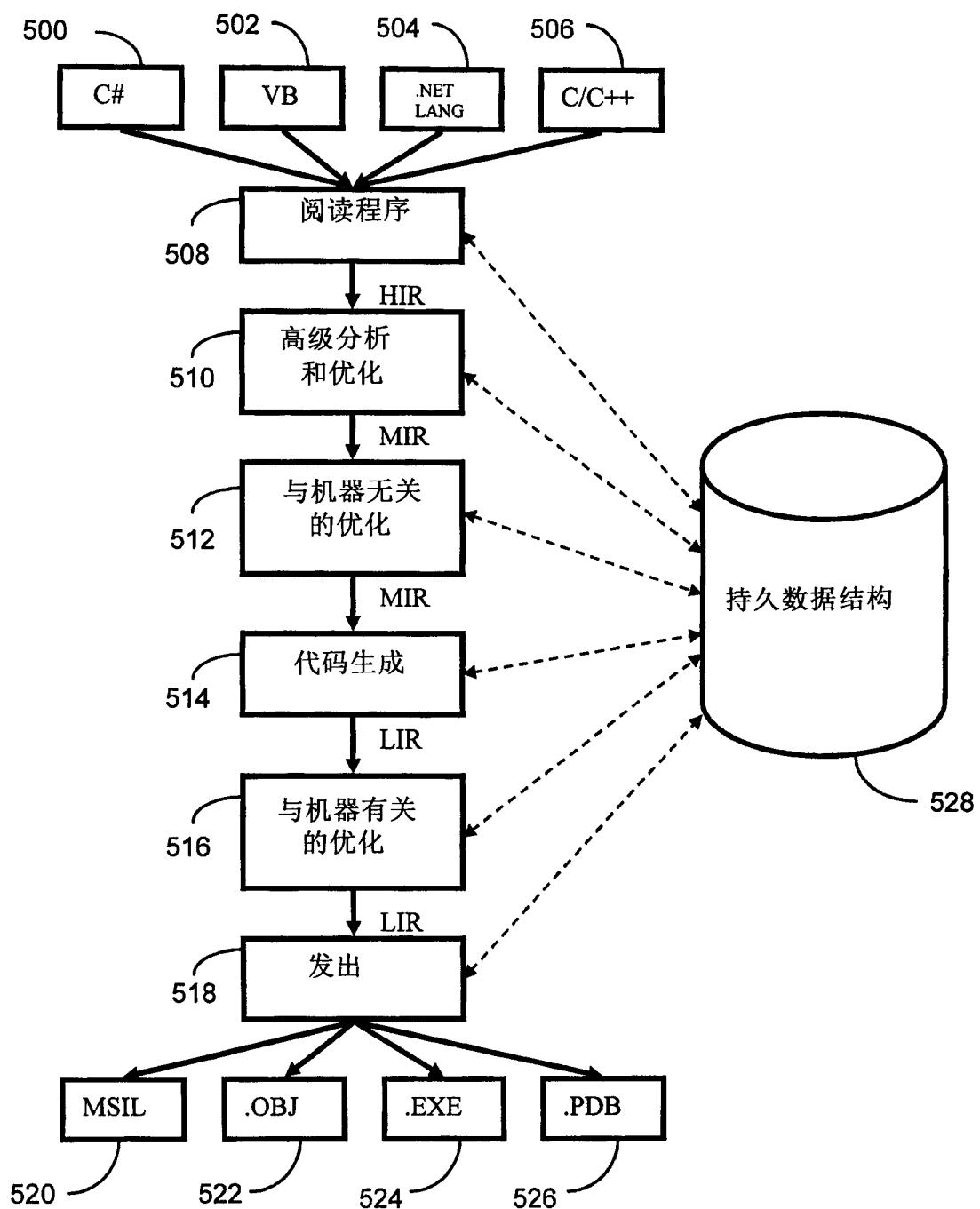


图 5

## 源

```

int foo(int a, int b)
{
    int r;

    if (a < b)
    {
        r = a + 1;
    }
    else
    {
        r = b + 1;
    }

    return r;
}

```

图 6(a)

## 高级的、与机器无关的IR的转储

```

_a.i32, _b.i32      = ENTER _foo
_t107.cond          = CMP(LT) _a.i32, _b.i32
                      CBRANCH(LT) _t107.cond, L2, L1
L2:
_t108.i32           = ADD _a.i32, 1.i32
_r.i32              = ASSIGN _t108.i32
                      GOTO L3
L1:
_t109.i32           = ADD _b.i32, 1.i32
_r.i32              = ASSIGN _t109.i32
                      GOTO L3
L3:
RETURN _r.i32
GOTO L4
L4:
EXIT _foo

```

图 6(b)

### 高级的具有SSA的机器无关的IR的转储

```
Explicit wiring of SSA graph using definition numbers shown in <#> blue.

<1>_a.i32, <2>_b.i32      = ENTER _foo                                #4
<3>t107.cond                 = CMP(LT) <1>_a.i32, <2>_b.i32
                               CBRANCH(LT) <3>t107.cond, L2, L1      #7
L2:
<4>t108.i32                  = ADD <1>_a.i32, 1.i32                #9
<5>_r.i32                     = ASSIGN <4>t108.i32
                               GOTO L3                         #9
L1:
<6>t109.i32                  = ADD <2>_b.i32, 1.i32                #13
<7>_r.i32                     = ASSIGN <6>t109.i32
                               GOTO L3                         #13
L3:
<8>_r.i32                     = PHI <5>_r.i32, <7>_r.i32
                               RETURN <8>_r.i32                #16
                               GOTO L4                         #16
L4:
                               EXIT _foo                      #16
                                         #17
```

图 6(c)

### 低级的、与机器有关的IR的转储(目标X86)

```
_a.i32, _b.i32      = ENTER _foo                                #4
{ESP}               = push EBP.i32, {ESP}                         #4
EBP.i32            = mov ESP.i32                           #4
ESP.up32->unk, EFLAGS.cc32 = sub ESP.up32->unk, 4.i32      #4
                       PROLOGEND                         #4
t110(EAX).i32     = mov _b[EBP.up32->unk].i32.a32          #7
t107(EFLAGS).cond = cmp(LT) _a[EBP.up32->unk].i32.a32, t110(EAX).i32  #7
                       jge(GE) t107(EFLAGS).cond, L1      #7
L2:
tv108-(EAX).i32   = mov 1.i32                           #9
tv108-(EAX).i32, EFLAGS.cc32 = add tv108-(EAX).i32, _a[EBP.up32-
>unk].i32.a32
#9
_r[EBP.up32->unk].i32.a32 = mov tv108-(EAX).i32           #9
                           jmp L3                         #11
L1:
tv109-(EAX).i32   = mov 1.i32                           #7
tv109-(EAX).i32, EFLAGS.cc32 = add tv109-(EAX).i32, _b[EBP.up32-
>unk].i32.a32
#13
_r[EBP.up32->unk].i32.a32 = mov tv109-(EAX).i32           #13
L3:
t113(EAX).i32     = mov _r[EBP.up32->unk].i32.a32          #16
L4:
                           EPILOGSTART                         #17
ESP.i32            = mov EBP.i32                           #17
EBP.i32, {ESP}     = pop {ESP}                            #17
{ESP}              = ret {ESP}                            #17
                           EXIT _foo, t113(EAX).i32          #17
```

图 6(d)

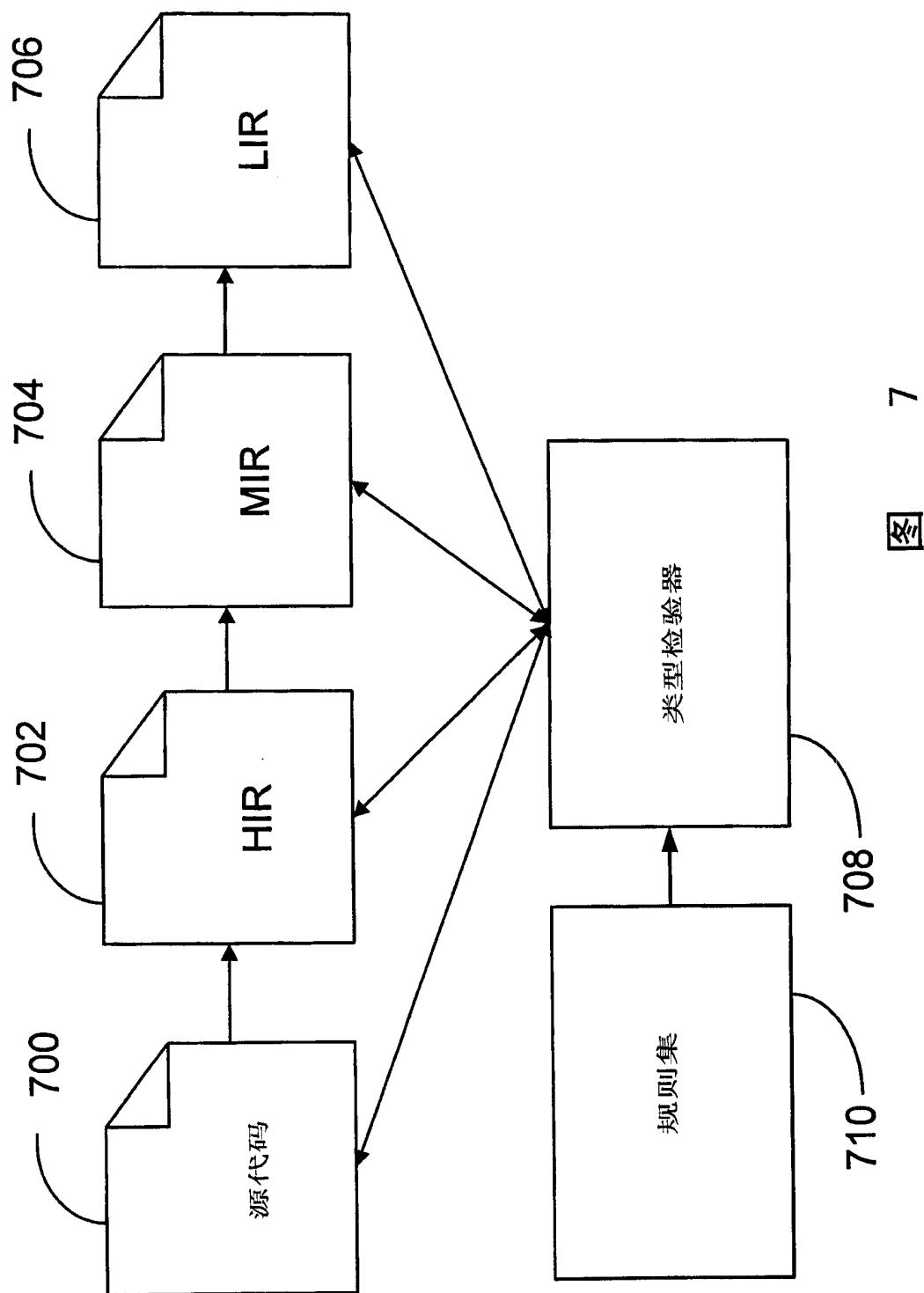
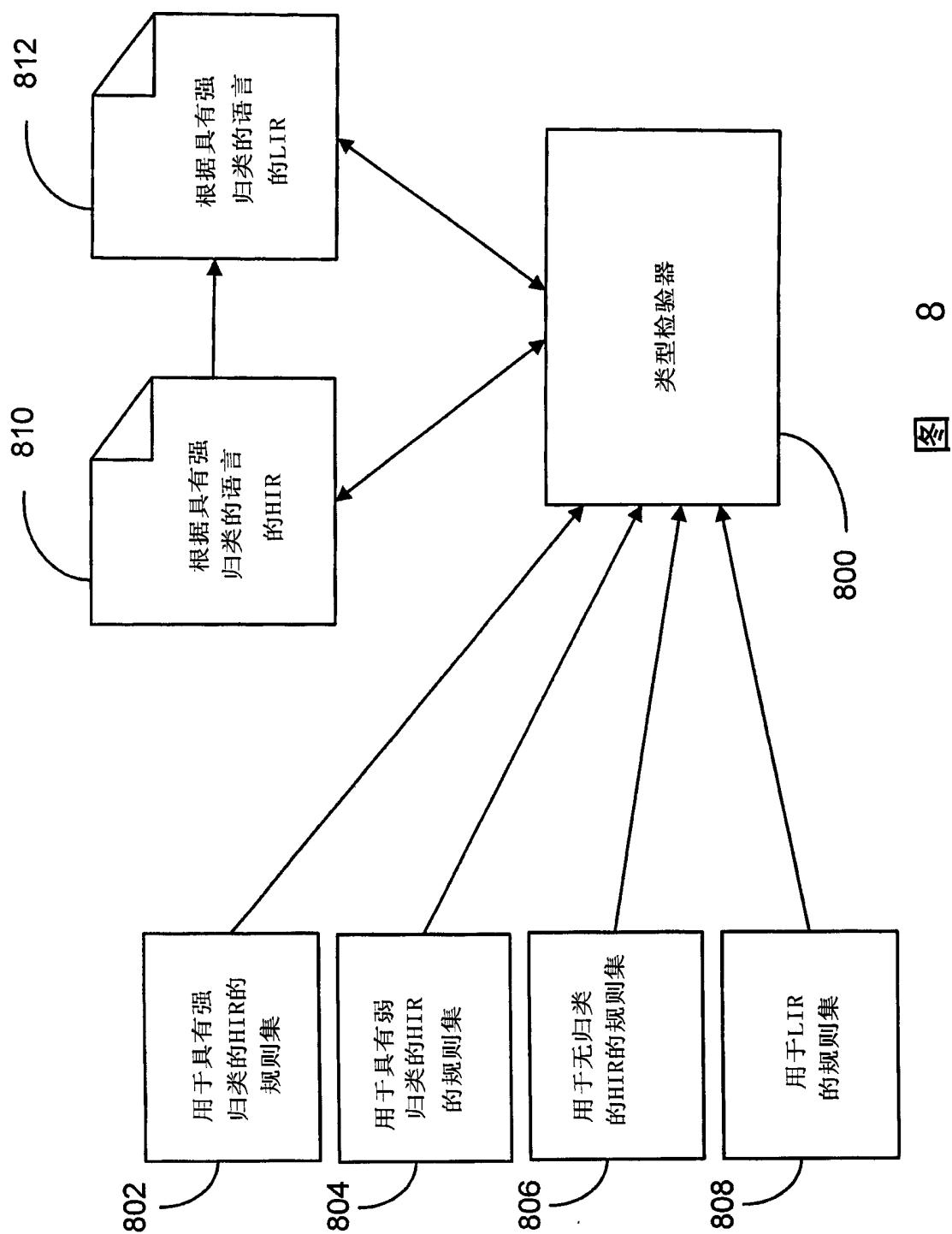


图 7



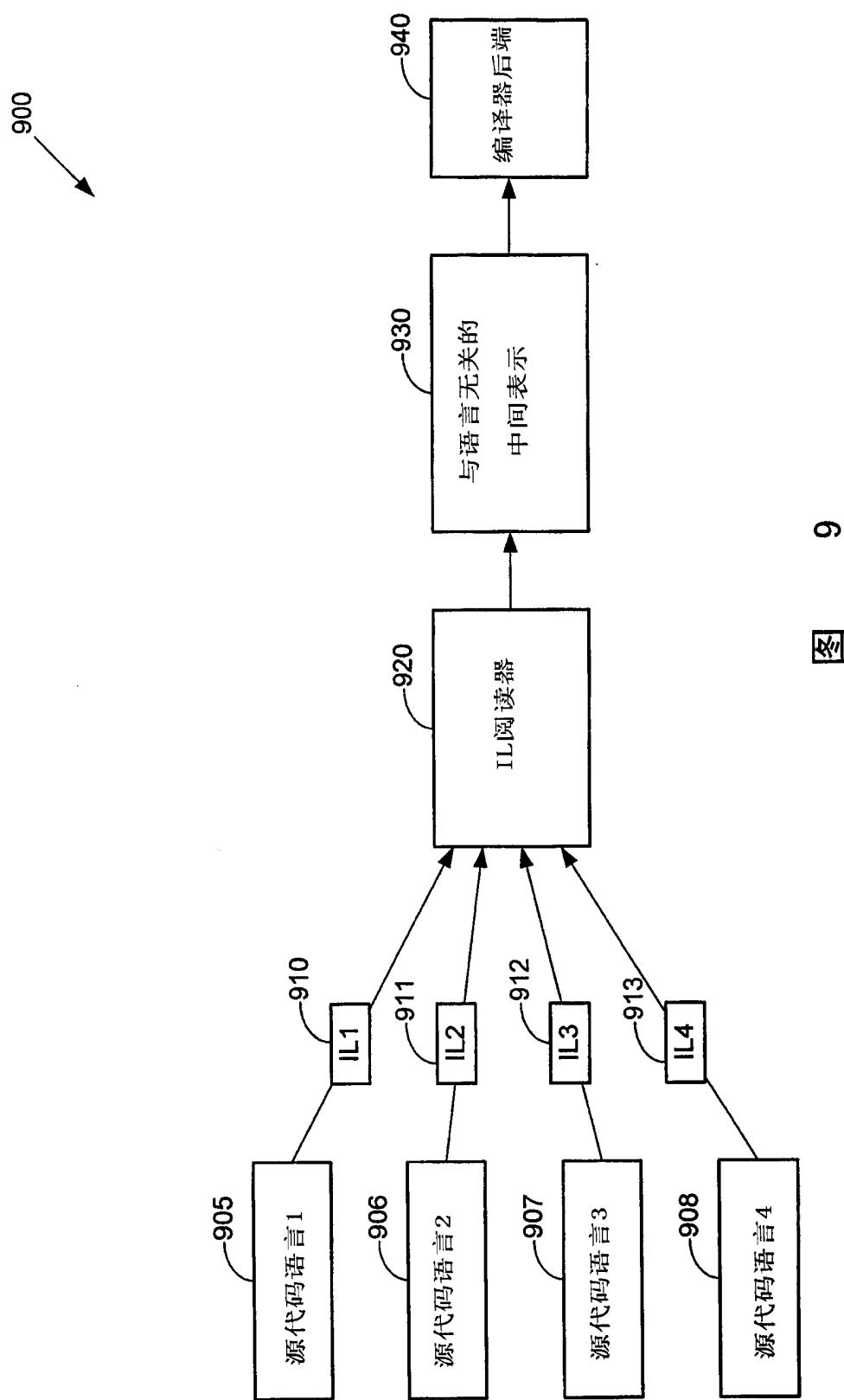


图 9

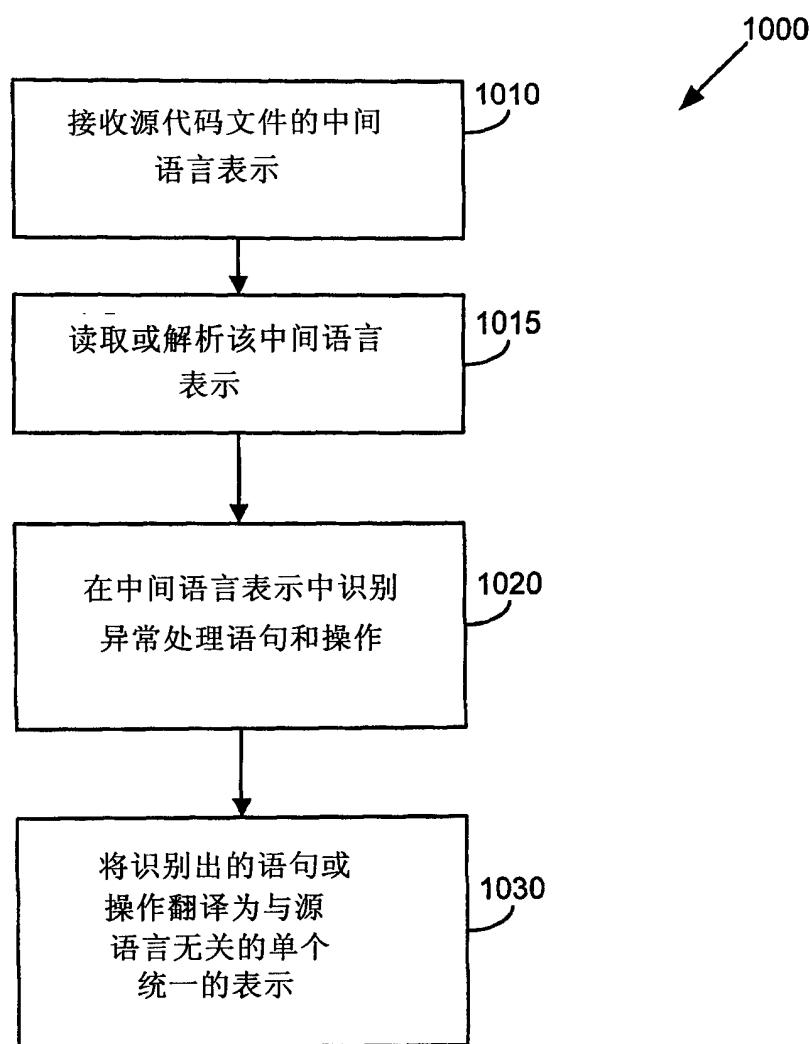


图 10A

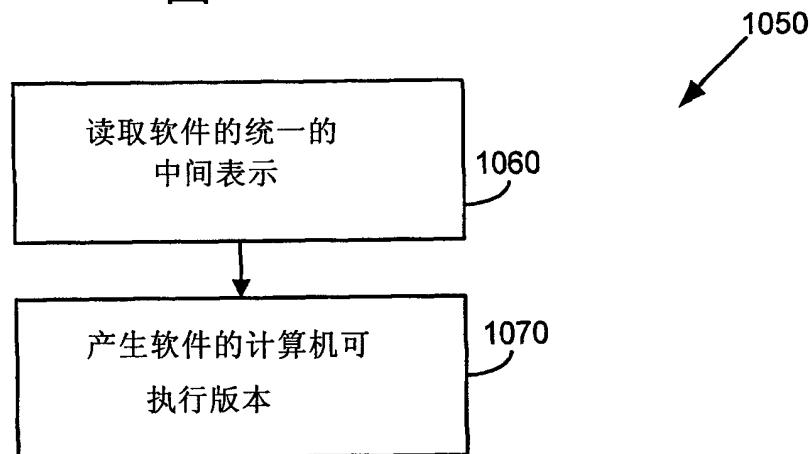
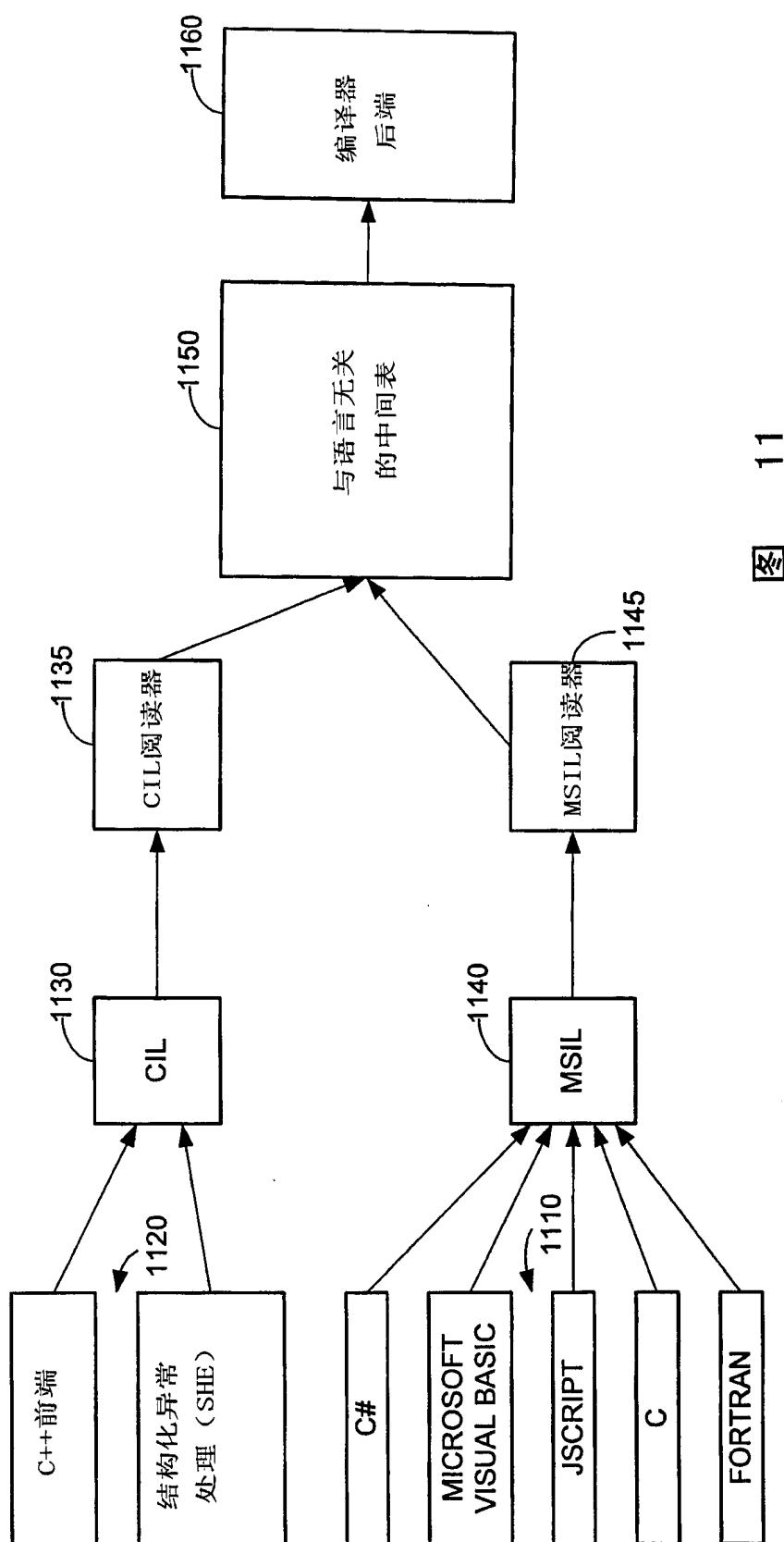


图 10B



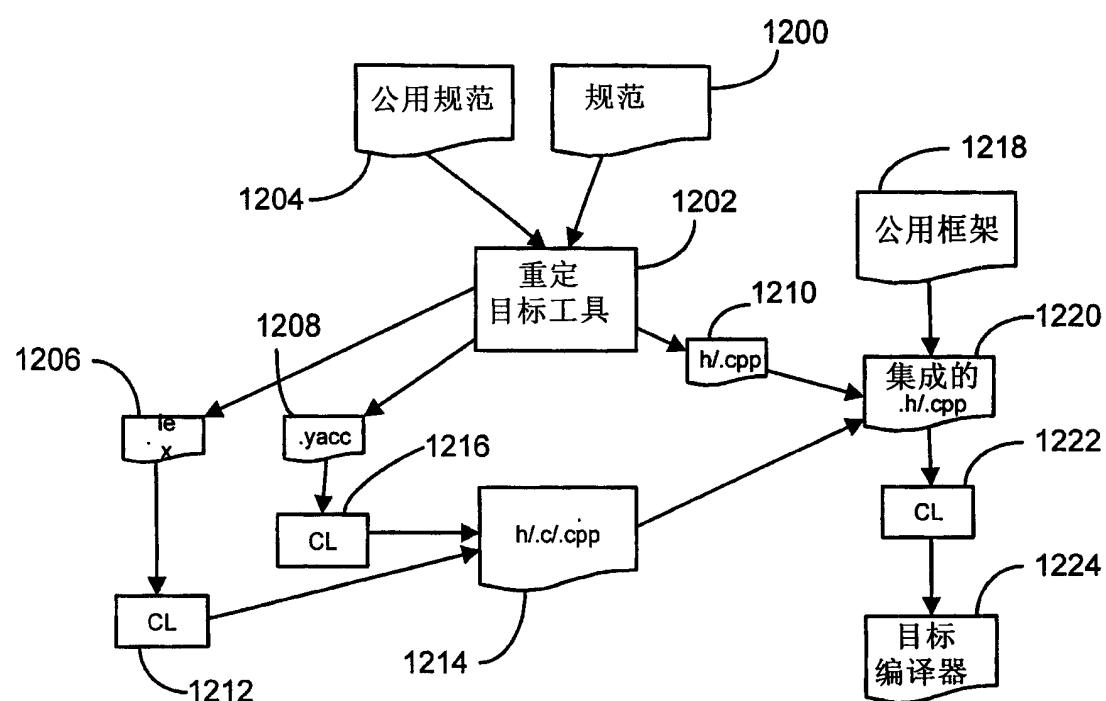


图 12

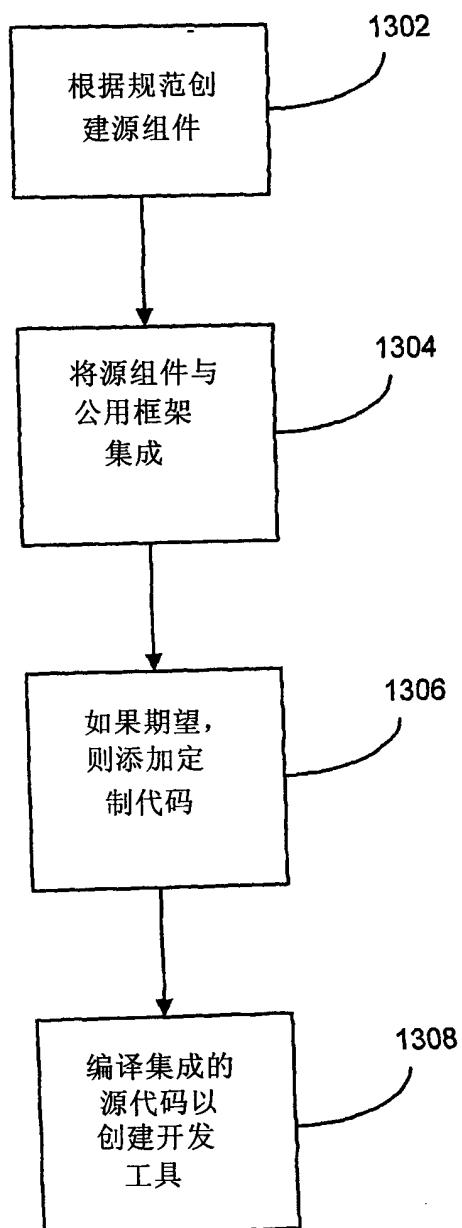


图 13

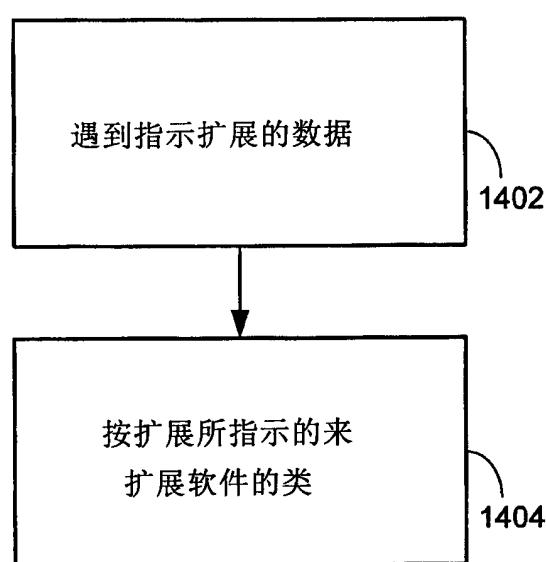


图 14(a)

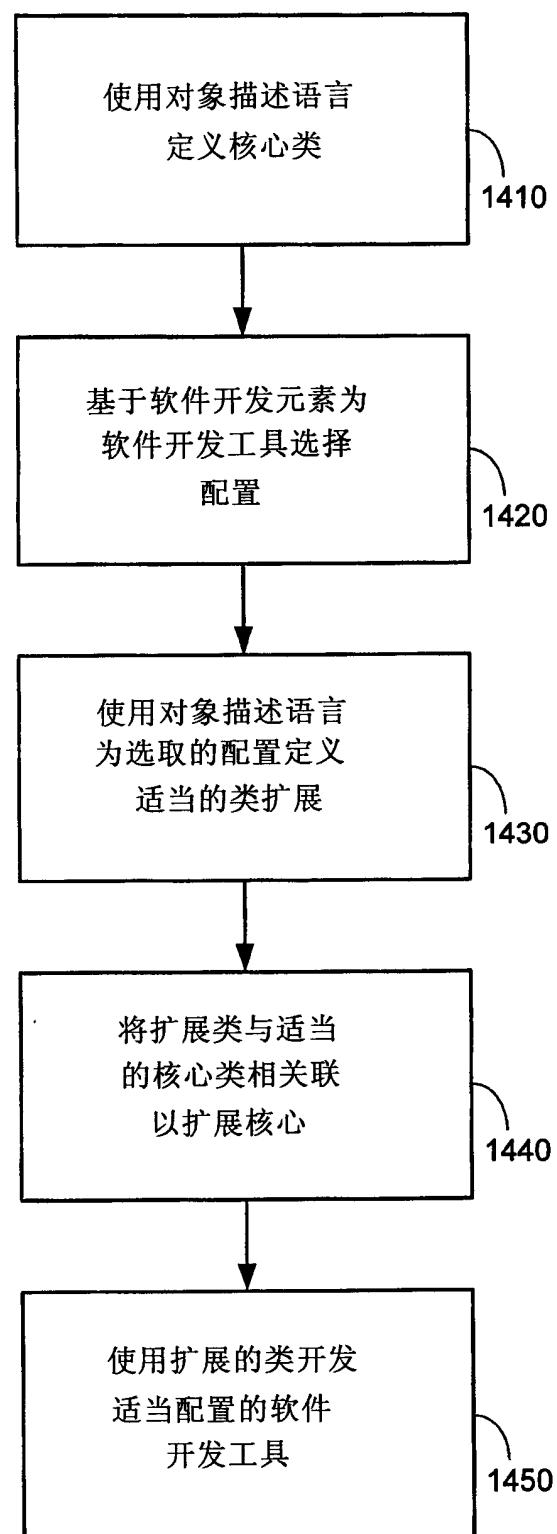


图 14 (b)

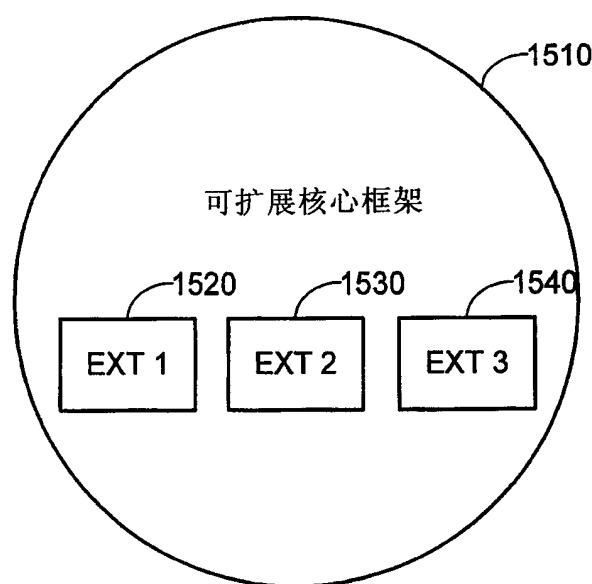


图 15(a)

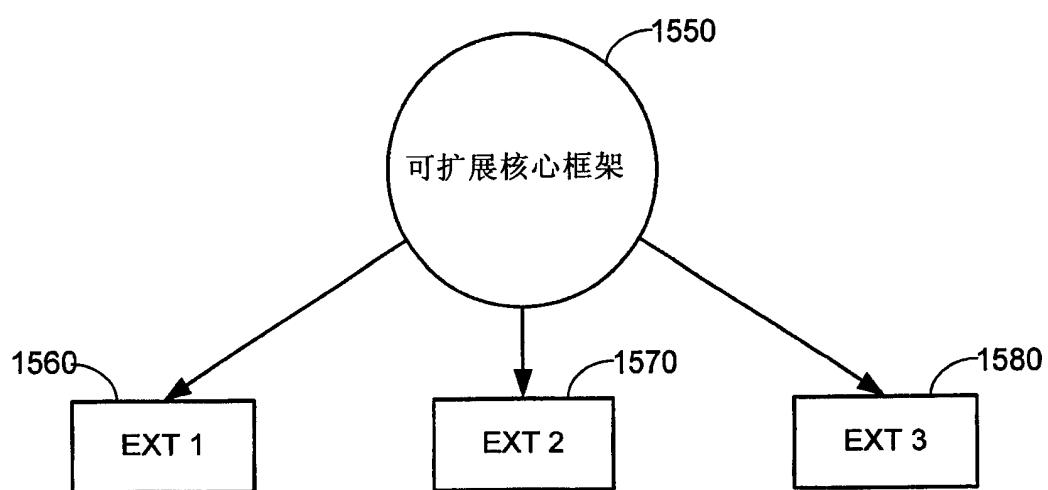


图 15(b)

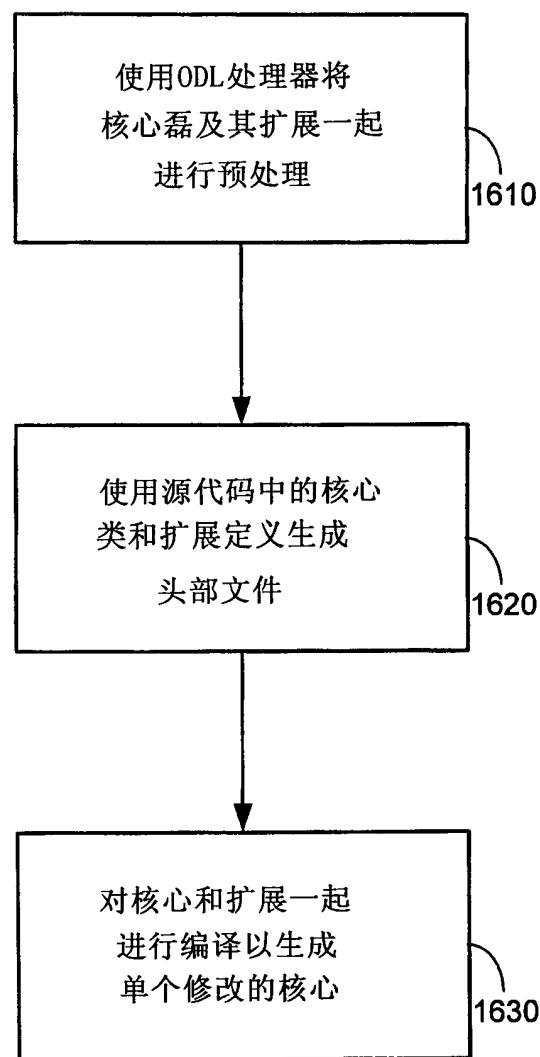
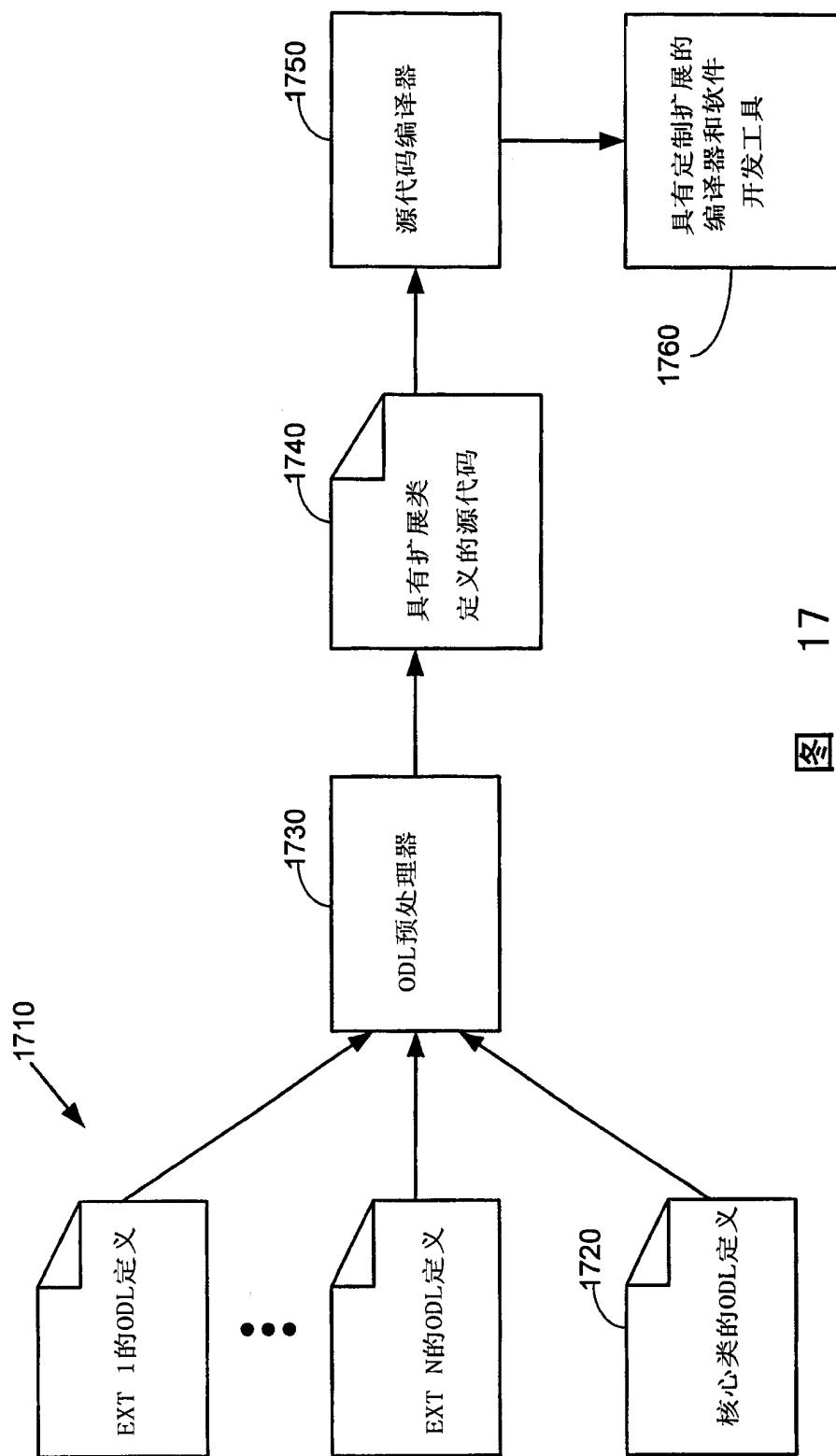


图 16



17

图

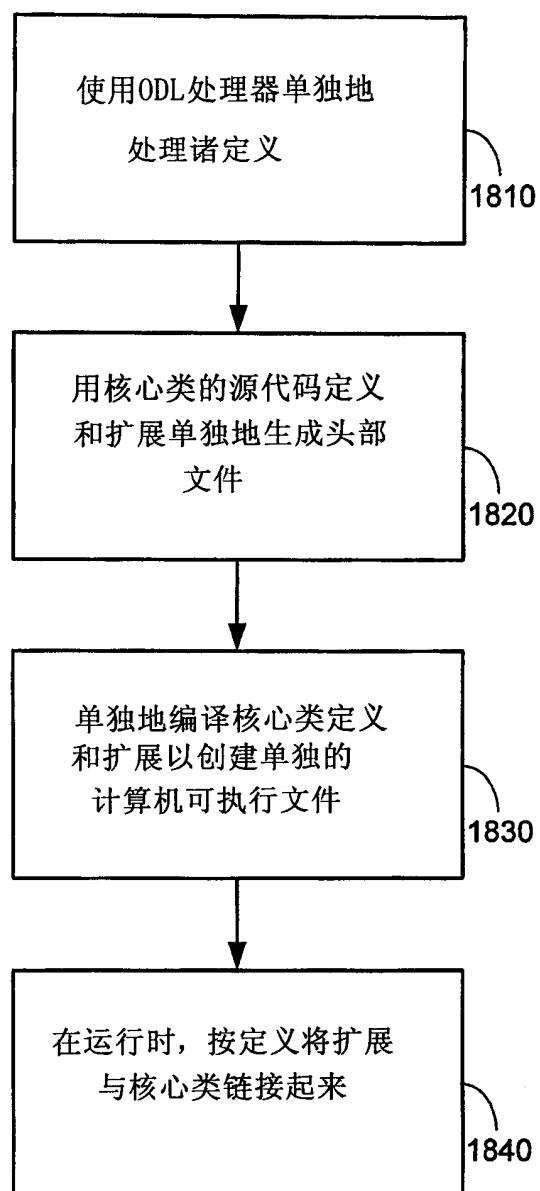
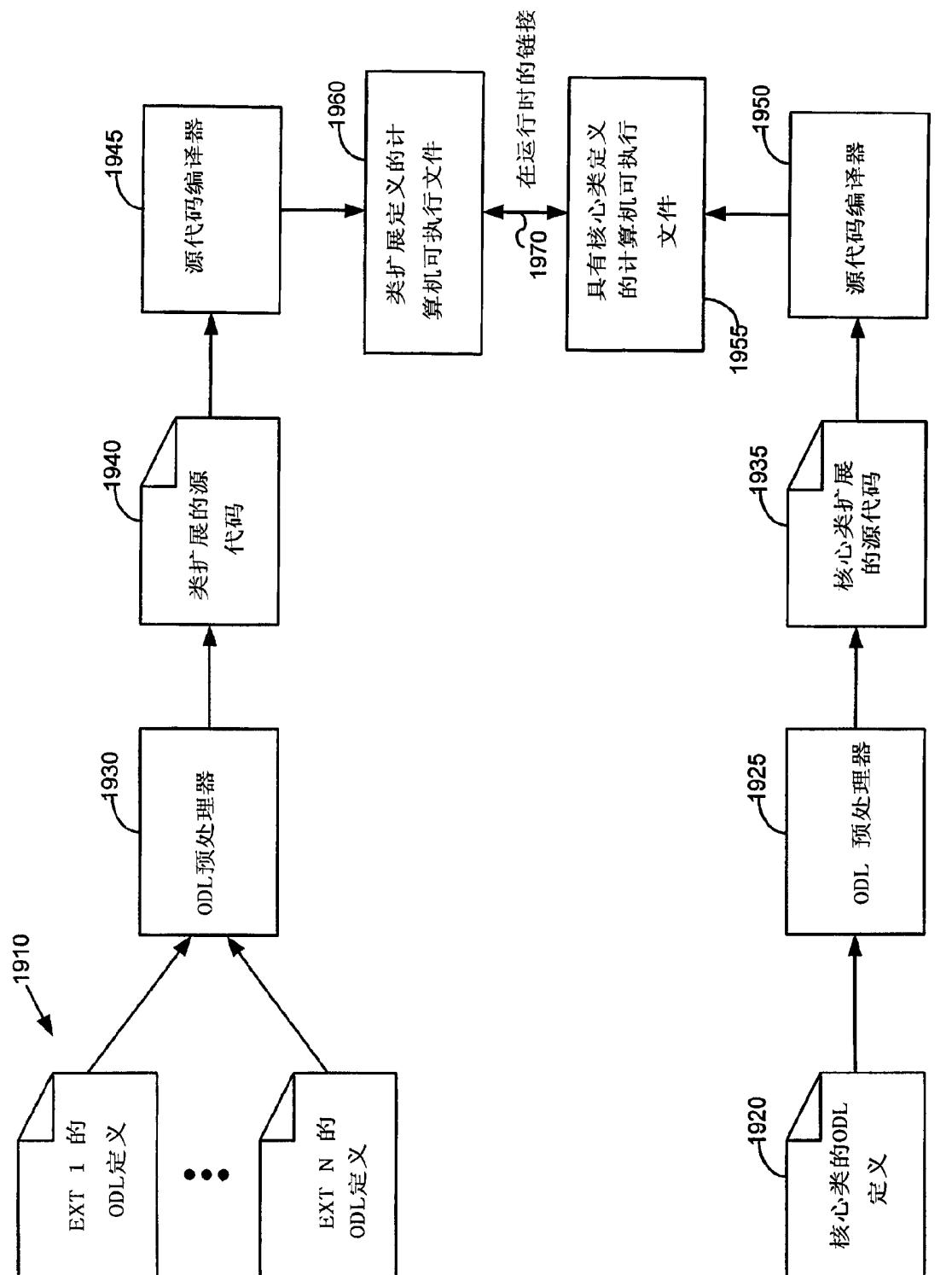


图 18



19

图

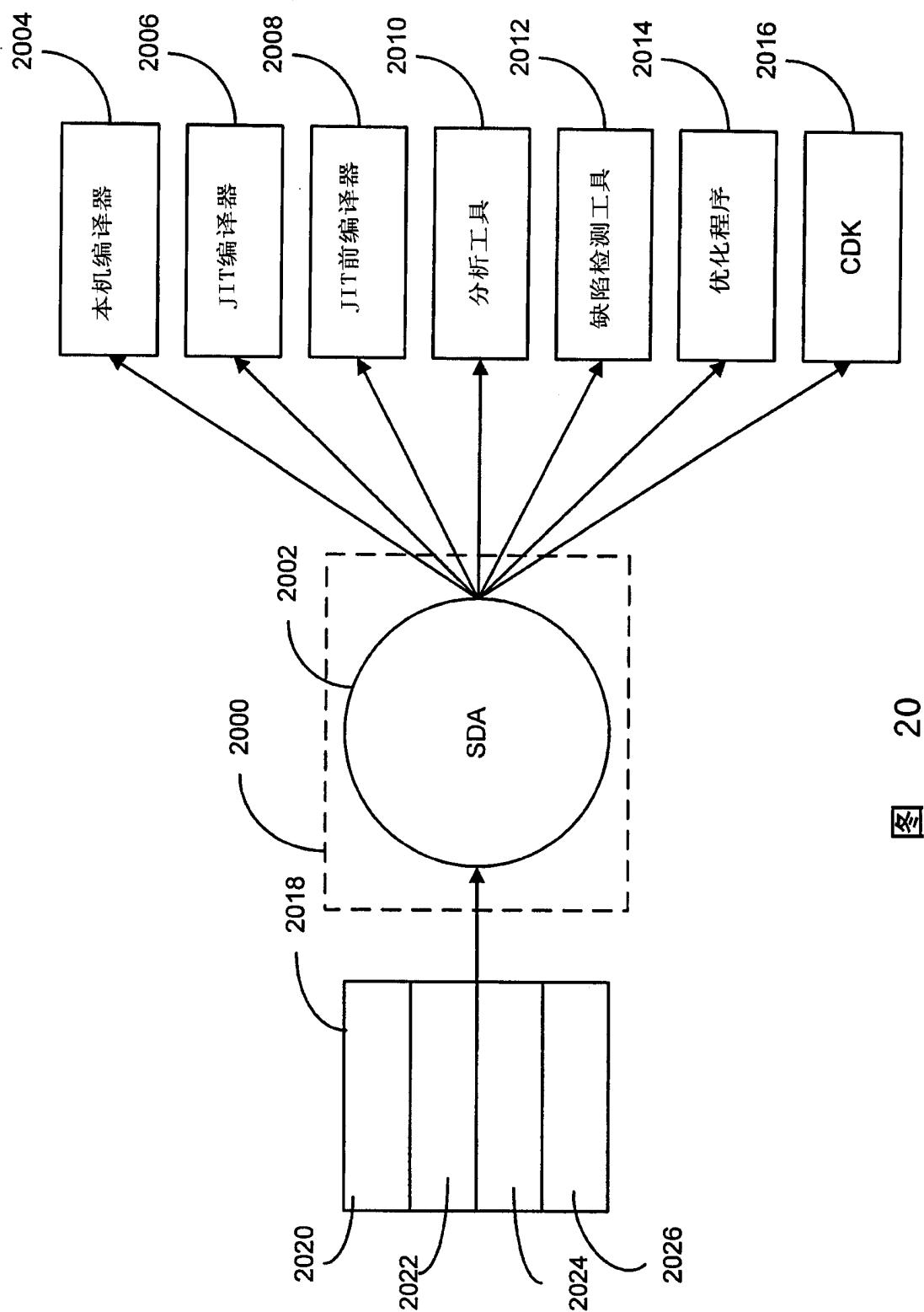


图 20

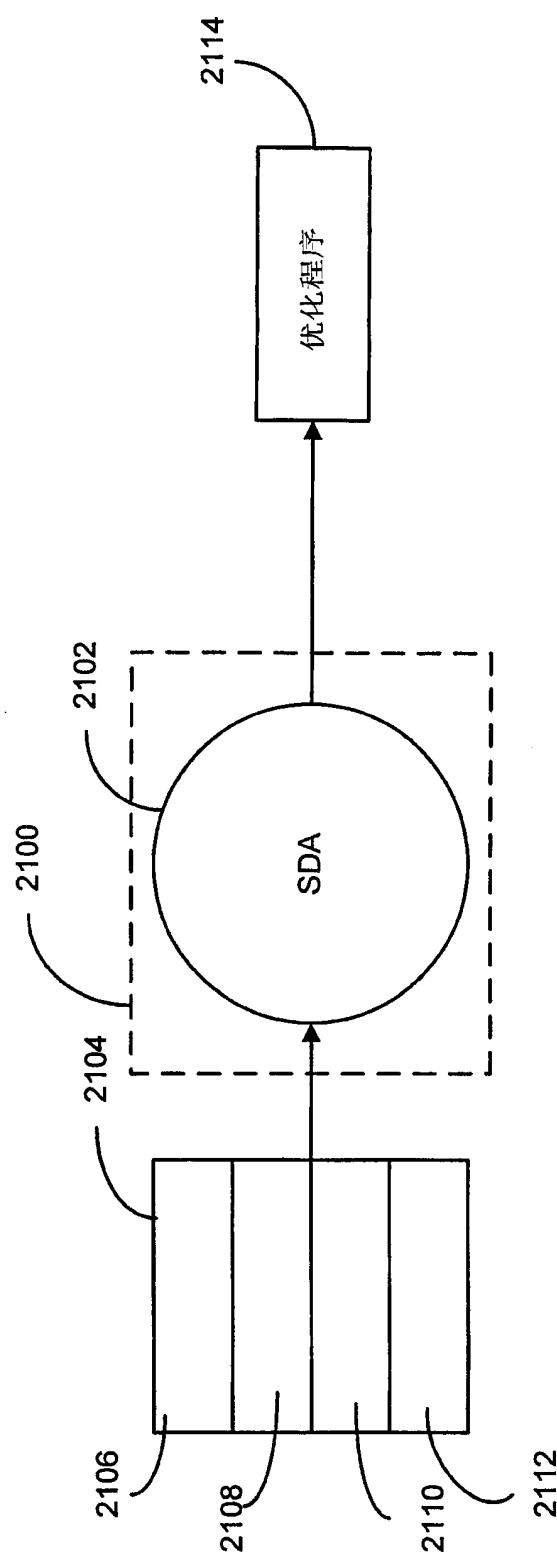


图 21

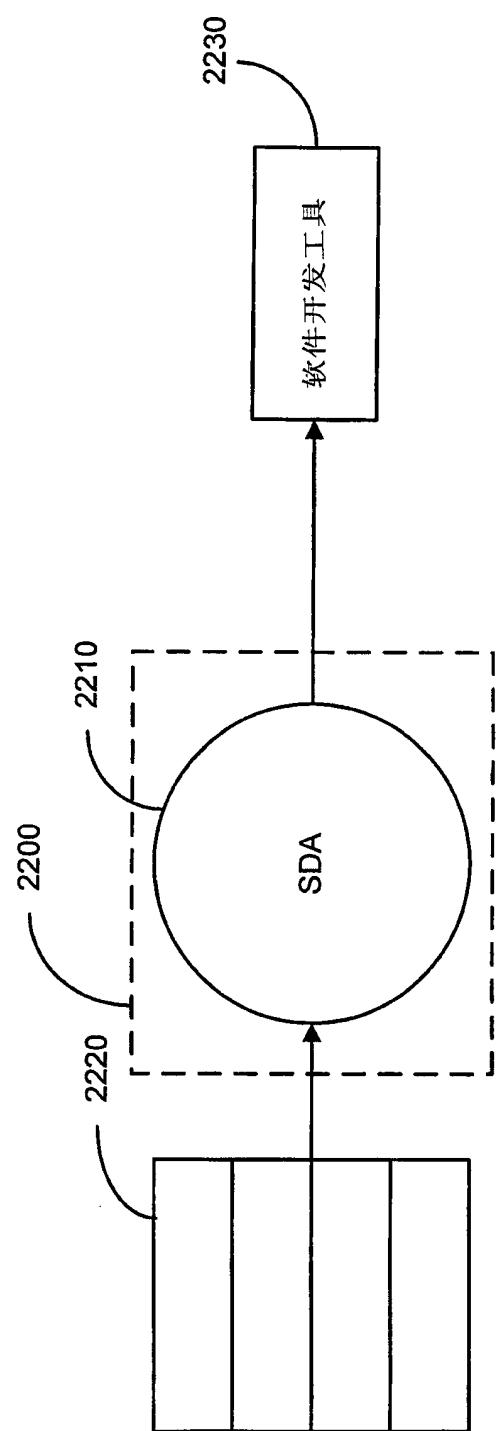


图 22

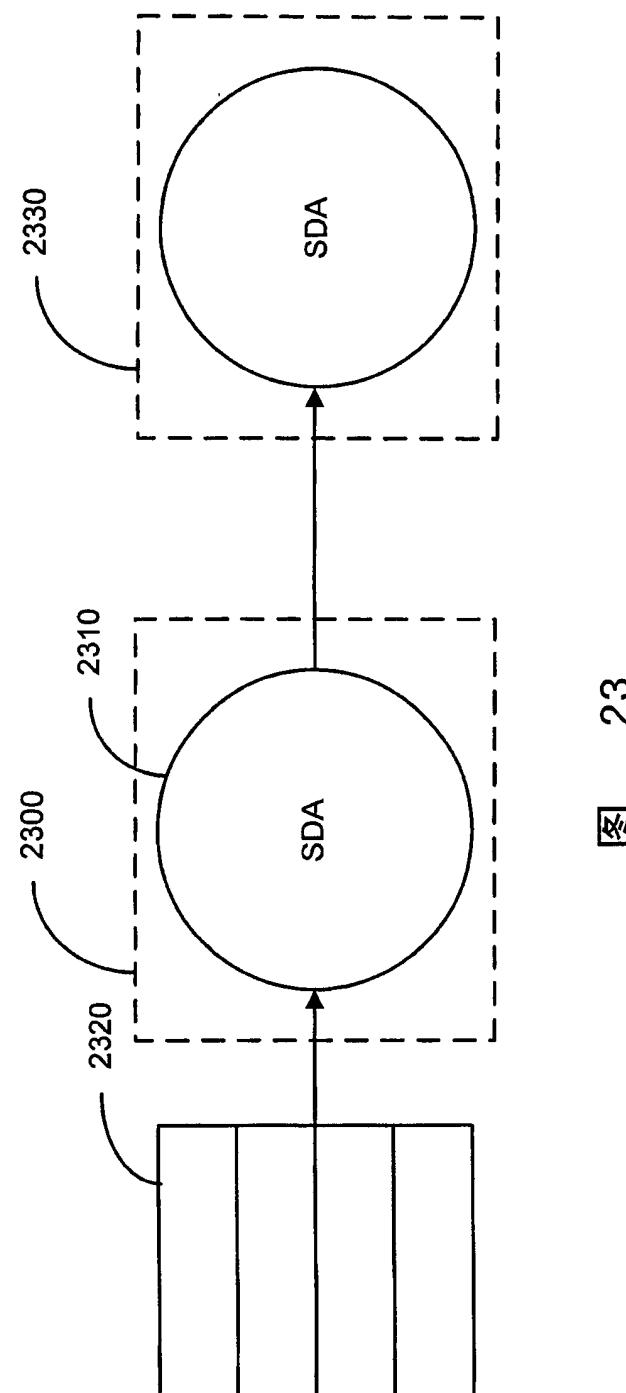
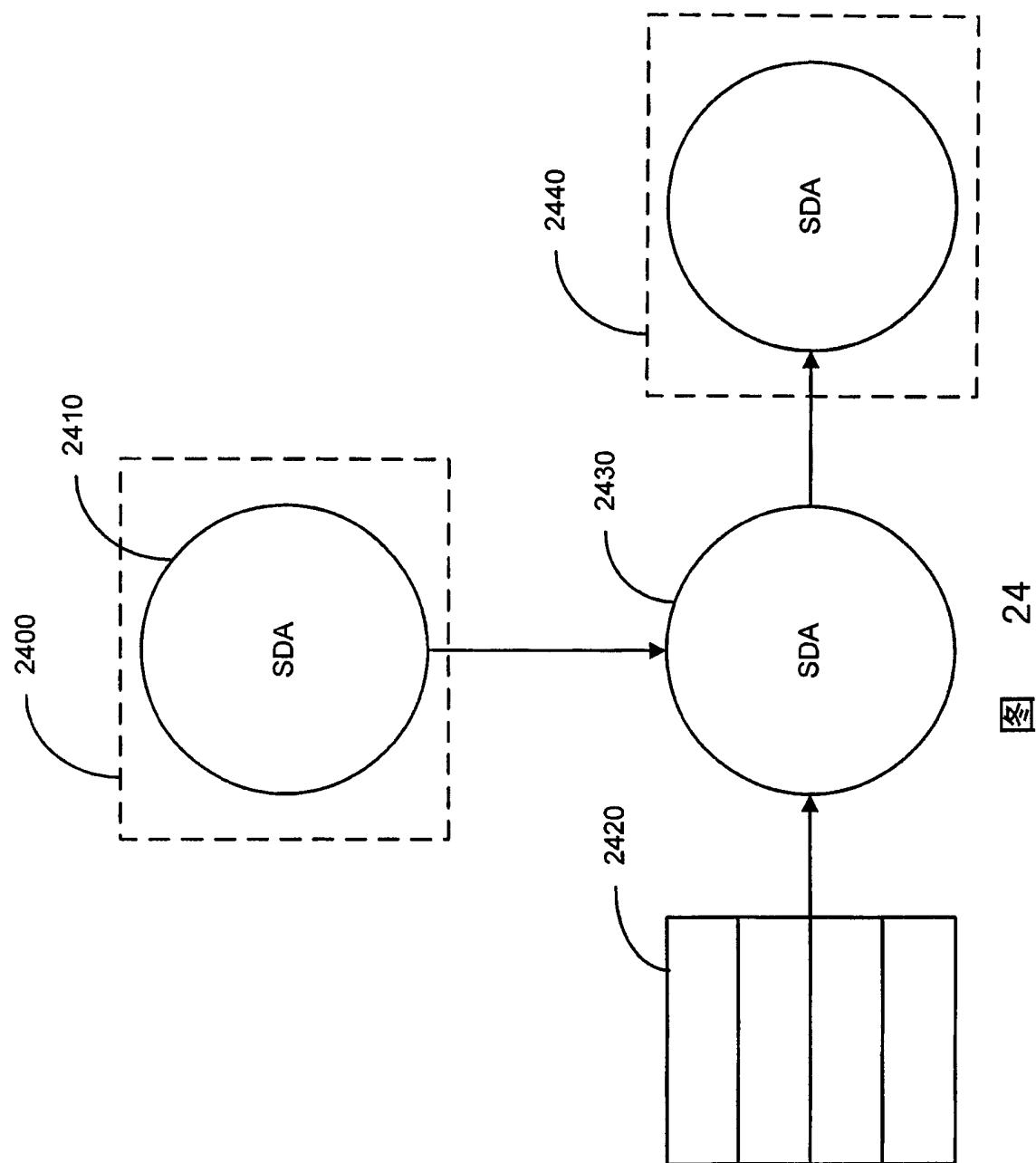


图 23



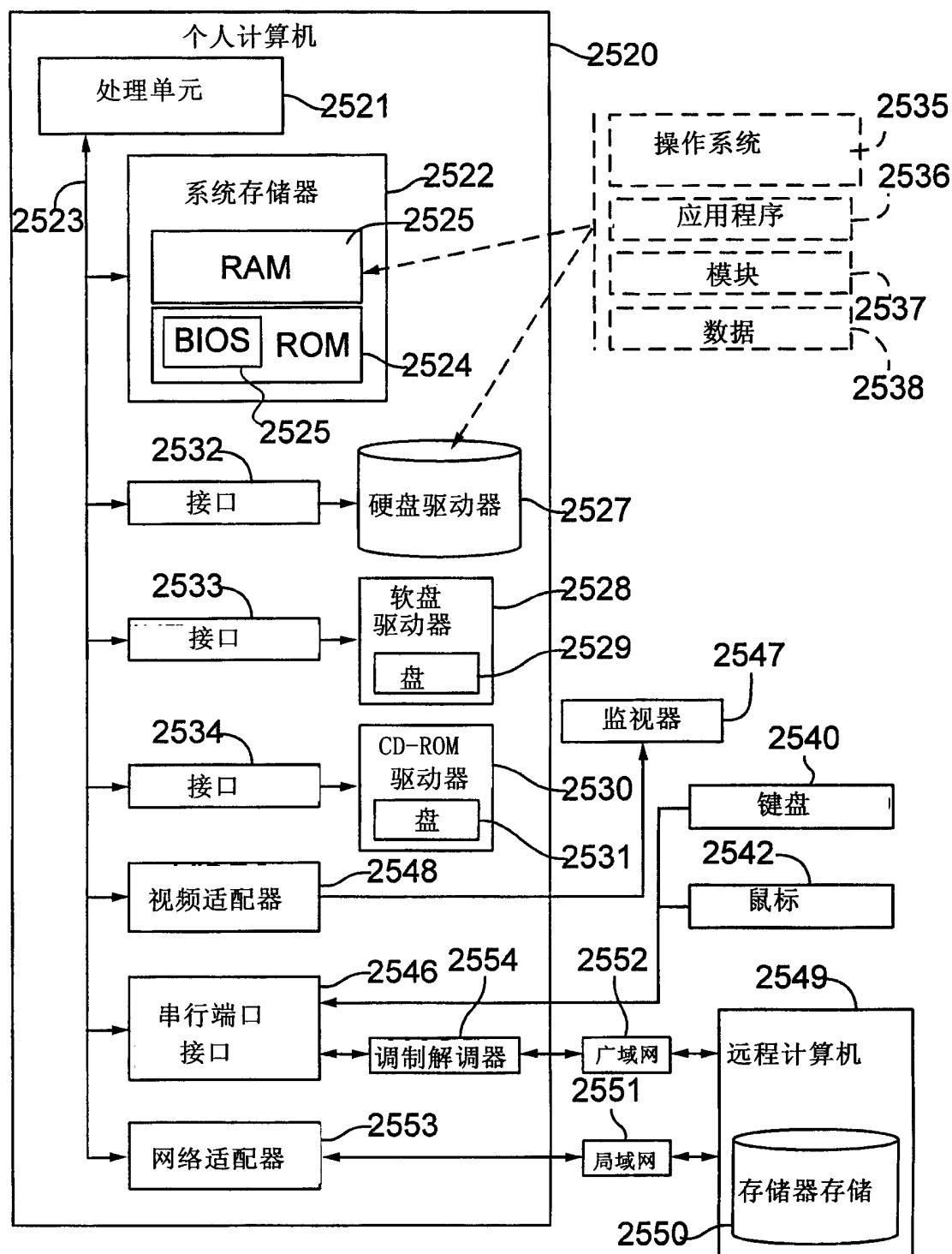


图 25