SIMPLE:一种新型多范型程序设计语言

吴 迪 陈 林 徐宝文

(南京大学软件新技术国家重点实验室 南京 210046) (南京大学计算机科学与技术系 南京 210046)

摘 要 为了满足越来越高的软件开发需求,许多通用程序设计语言扩充了各种新的语言设施,从而使语言变得复杂而难于学习和使用。为了创造一个核心概念简单明确、同时可以广泛用于各类开发的语言,设计了一种具有简明核心概念和丰富语言设施的程序设计语言SIMPLE。首先对SIMPLE语言进行概述,然后针对SIMPLE的模块化、泛型、内存管理以及异常处理等设施进行阐述。此外,讨论了如何将过程式、面向对象、函数式3种程序设计范型在SIMPLE中实现有机的融合。

关键词 程序设计语言,语言设计,多范型程序设计,面向过程程序设计,面向对象程序设计,函数式程序设计

中图法分类号 TP312 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.07.001

SIMPLE:A Novel Multi-paradigm Programming Language

WU Di CHEN Lin XU Bao-wen

(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210046, China)

(Department of Computer Science and Technology, Nanjing University, Nanjing 210046, China)

Abstract Because modern static languages become increasingly complicated with too many extended constructs, we wanted a language with simple core concepts and wide applications. Therefore, we designed SIMPLE, a high-level programming language that possesses concise key concepts and abundant language constructs. We first introduced SIMPLE in a nutshell. Then some illuminating ideas on modular programming, generic programming, garbage collection, and exception handling were proposed. In addition, we discussed how to integrate procedure oriented programming, object oriented programming, functional programming paradigms in the design of SIMPLE.

Keywords Programming language, Language design, Multi-paradigm programming, Procedure oriented programming, Object oriented programming, Functional programming

1 引言

如今,软件系统正变得越来越复杂,但是能够在诞生之初就适用于各种软件开发环境的设计的语言却很少。为了在不同的程序设计环境中得到广泛应用,许多高级程序设计语言通过不断扩充语言设施来增强自身功能。然而,语言功能的扩展往往导致语言框架变得复杂与庞大,甚至与最初的语言设计产生不一致。比如,作为20世纪80到90年代最具代表性的语言之一,Ada由美国国防部主持,汇集当时全球最优秀的软件工程领域的专家设计而产生。最初版本的语言设计(Ada 83)体现了良好的模块化软件开发思想,在当时为规范软件项目开发提供了有力的支持。但随着软件开发复杂度的提升以及以面向对象为代表的新一代程序设计范型的广泛应用,Ada逐渐被淘汰。为了使语言重获生机,Ada研究者们不断为其增加新型的、在其他流行语言中得到广泛应用的语言设施,最新的Ada 2005[12]以及Ada 2012[17]版本将大量面向对象和并发程序的设施作为Ada语言的核心组成部分。虽然这些新增的设施可以帮助Ada语言更适用于多种领域软件的开发,但是与Ada 83、Ada 95相比,新增加的语言设施破坏了Ada原有的完美核心框架。

与Ada的情况类似,由于最初设计的局限性,许多其他语言通过不断增补语言设施来实现广泛的应用,但是添加的设施往往会影响原先语言核心框架的设计,从而不仅破坏了语言最初设计的美感,而且增加了用户尤其是初学者对语言理解和使用的难度。

纵观程序设计语言的发展,我们发现优秀的语言往往具备简单而稳定的语言核心,并且拥有良好的可扩展性。核心框架的简单稳定可以保障语言易于被用户接受,而良好的可扩展性则有利于语言的发展,使语言设施的更新不破坏语言的核心框架。Java与C#之所以能够在过去十几年间在程序设计语言界占据一席之地,从技术角度而言,是因为它们优秀的设计框架和良好的扩展性为其提供了重要支持。首先,Java与C#均为面向对象语言,且最初的语言设计充分体现了面向对象特性,因此在当今以面向对象范型为主流的环境中,

到稿日期:2014-04-08 返修日期:2014-05-10 本文受国家重点基础研究发展计划项目(2014CB340702),国家自然科学基金项目(61170071),江苏省自然科学基金(BK2011190)资助。

吴 迪(1988-),男,博士生,主要研究方向为程序设计语言,E-mail:NJU_wudi@163.com;陈 林(1979-),男,博士,讲师,主要研究方向为软件分析与测试、软件重构、错误定位;徐宝文(1961-),男,博士,教授,博士生导师,主要研究方向为程序设计语言、软件工程、并行与网络软件。

它们的语言核心框架无需做出重大变革。其次，为了满足不同程序设计环境的需求，Java 与 C# 均补充了新的语言设施（比如，Java 5.0[26]版本开始支持泛型程序设计，C# 3.0[27]版本增加了部分函数式程序设计的设施），但是增加的设施作为可选的部分，并未破坏原有的语言框架，同时，以往语言版本的语法在新的版本中得以完整的保留。

虽然以 Java、C# 等为代表的通用程序设计语言在语言设计及应用方面取得了巨大成功，但是单一的程序设计范型无法适用于所有软件开发的环境（比如，过程式程序设计更适用于自顶向下、逐步求精的开发环境，而函数式程序设计更适用于理论证明和公式推导）。我们认为，一个良好的、通用的语言应当具备若干种不同的程序设计范型，以辅助用户在面对不同种类的问题时选择不同的思维方式和解决方案。

因此，我们希望创造一种既具备简单而稳定的语言核心又包含多种常用程序设计范型的语言。在这种理念的指引下，我们设计了 SIMPLE 语言。SIMPLE 借鉴了传统语言中优秀的设计理念，并融合了当今流行语言中成功的语言设施，它的核心框架简单明确，并且易于被用户接受。我们将简单性、可读性、可靠性、安全性、可扩展性、效率作为语言的基本目标。同时，我们尽量使各种语言成分在语法规则上保持一致，并在语义解释方面形成有机的整体。此外，过程式、面向对象、函数式程序设计范型的结合使 SIMPLE 为用户提供了在面对各式问题时不同的解决之道。

本文第 2 节介绍 SIMPLE 的设计目标；第 3 节阐述 SIMPLE 的典型特征并通过示例和比较给出说明；第 4 节描述多范型程序设计在 SIMPLE 中的体现；第 5 节对 SIMPLE 的设计方案进行总结并给出语言文本手册的下载地址。

2 设计目标

程序设计语言的评估标准是计算机科学工作者一直争论的问题，不同领域的研究人员有着不同的看法，即便是相同领域的研究人员由于角色的不同也可能存有争议。比如，软件设计者往往希望所使用的语言具有简单性与优秀的可扩展性；软件维护者着重关注语言的可读性与可维护性；用户则要求语言具有较高的可靠性与效率[30]。

综合软件开发过程中不同角色所关注的语言评估标准，SIMPLE 选取了其中最重要的 5 个作为语言的基本设计目标，即简单性、可读性、可靠性、安全性、可扩展性以及效率。尽管这些标准已经在诸多程序设计语言中得以体现，但是 SIMPLE 在设计过程中将这些标准作为整体进行考虑，并且使它们在语言设计的每一个环节得以体现。

2.1 简单性

多样化的语言设施有利于提升编程效率，但是同时也加重了程序员学习和使用语言的负担。如果程序员不熟悉自己所用语言中的所有设施，往往会导致一些语言特征被错误使用[1]。正如 Hoare 教授所说，程序员偶尔使用未知的语言特征也许会导致灾难性的后果，他建议语言中的设施应该尽量精简并通过设定一系列相容的规则使所有语言设施得到统一的组织和管理，这样可以帮助程序员减少误用程序特征的几率[1,2]。

顾名思义，SIMPLE 的字面意义是简单，因此简单性便成为 SIMPLE 的首要目标，简单性是指语言中的概念与成分简单、数目少，不包括功能重复或相似的概念变种，使语言具有良好的正交性。SIMPLE 的简单性（simplicity）体现于所有语言成分概念简明，且在形式上能与其他语言成分相协调，程序员在理解这些概念时不会产生歧义。

为了能实现简单性，使语言核心框架协调统一，同时保证所有语言设施功能明确，SIMPLE 对不同语言设施采用相似的语法描述，即，使用不同语言设施编程时所使用的语法是类似的。通过这种方法，程序员（尤其初学者）可以轻松掌握语言设施的使用方法，从而使编程时出现的语法错误得到有效控制。

不同语言设施在语法规则上的统一不仅有利于程序员对语言的掌握和使用，而且有利于编译器在词法分析阶段对各种语言设施以相似的方法进行处理。

2.2 可读性

可读性（readability）是指阅读使用某种语言编写程序的方便程度。在软件完成编码工作后，通常需要进行若干轮调试与测试，这意味着程序会经历一次编写和多次阅读，因此语言的可读性至关重要，它也是保障软件可维护性的基础，Ada 等语言将可读性作为重要的设计目标之一。

SIMPLE 从传统程序设计语言（Ada、Modula 等）中借鉴了一系列技术用于提升程序的可读性：

- 良好的模块化特性。众所周知，模块化意味着程序可以分为若干个独立的单元进行编码和编译，并通过合理的机制确保模块间的一致性。SIMPLE 提供了若干种不同粒度的程序单元用于支持模块化程序设计，比如类（class）用于组织抽象数据类型及相关操作，模块（module）用于构建结构更复杂、功能更丰富的数据单元。用户可以把一组有关的类型、数据、子程序、异常等信息封装到一个模块中，以保持与其他实体的相对独立性，通过对信息的抽象与封装提高了程序的可读性。

- 规格说明与体相分离。程序单元的规格说明类似于接口，体类似接口的实现。规格说明与体相分离的技术最早出现在 Ada 语言中，集中体现了信息隐藏原则。在团队软件开发中，测试和维护人员只需要关注每个单元的规格说明，而无需理解单元内部的具体实现。这种技术不仅有利于程序组织的更加清晰，而且减轻了测试和维护人员的负担。

- 关键字括号结构。现代流行的程序设计语言基本采用花括号结构来表示程序块，但是在多层嵌套的程序结构中，如果出现括号不匹配的情况，那么在检查缺失的括号位置时往往会耗费程序员大量精力。关键字括号结构要求每个块结构均由一对关键字进行约束，从而有效规避了花括号结构所带来的问题。

2.3 可靠性

如果一个程序在任何条件下的运行都能达到它的说明标准，那么我们称这个程序是可靠的[1]。可靠性（reliability）在高级语言（特别是军用高级语言）中占有重要地位，曾由美国国防部发起设计的 Ada 语言便将可靠性与可维护性作为设计的首要目标。SIMPLE 在设计过程中充分关注各种语言成分的可靠性，以下是几种用于保障程序可靠性的设计策略：

- 静态类型检查。SIMPLE 采用强类型来确保程序的类型安全性，即：1. 程序中的所有对象必须先声明后使用；2. 对象在声明时必须指明类型；3. 对象的所有运算必须保持类型

不变，且得到指定类型的结果。强类型机制确保了类型检查的有效进行，SIMPLE 要求在编译时对所有变量及表达式都进行类型检查，以减少程序运行时的类型错误。

• 错误预防。SIMPLE 通过一些严格的语法形式保障程序员在编写程序时不会犯某些错误，例如，SIMPLE 要求 for、while、do-while 等循环语句的控制变量是局部于循环语句的，即在循环语句外部无法访问循环的控制变量，这就避免了像 C 程序中在循环外部随意修改循环变量的现象。此外，SIMPLE 强类型机制要求按名等价，不允许不同类型的对象之间互相赋值，以及不相容的形参和实参的参数匹配。这些方式迫使程序员显式说明其编程意图，使他们意识到程序中潜在的错误，从而减少错误出现的可能性，使程序更加可靠。

• 异常处理。程序若在执行中出现错误时能够及时改正并恢复执行，则将有助于提高其可靠性，这种异常处理机制已经广泛应用于高级语言中。SIMPLE 对单独异常处理机制（Ada 所采用的异常处理机制）进行改进，通过将异常作为独立的程序单元并辅以异常处理过程的方式使得 SIMPLE 具有较强的异常处理和恢复能力，从而提高了程序的可靠性。

2.4 安全性

安全性（security）是指程序在遭到恶意外部攻击的情况下依然能够正确运行的能力。常见的威胁程序安全性的手段包括获取程序后门、拒绝服务、缓冲区溢出等，攻击者可以通过对内存的控制截取或篡改数据库等资源中的重要信息，因此语言良好的内存管理对提高程序安全性至关重要。为了有效防止常见的内存攻击，保障程序安全性，SIMPLE 提供了良好的内存管理方法：

• 指针封装。众所周知，像 C 语言一样将指针直接暴露给用户的方式往往导致内存泄露和空指针等现象，严重的指针错误会造成程序崩溃，而且黑客常常会利用程序中使用不当的指针获得对关键数据的控制权限，严重危害程序的安全。SIMPLE 对指针进行了封装，禁止将指针直接暴露给用户，只允许用户通过引用进行对象访问，而且 SIMPLE 对不同类别的对象采取不同的访问控制手段，进一步降低了攻击者通过内存访问破坏程序安全性的可能性。

• 显式区分内存空间。与 Java 将全部对象存储在堆空间不同，SIMPLE 严格区分栈对象与堆对象——栈对象由系统自动分配和回收，堆对象则允许通过引用进行操作。栈对外界的隐藏避免了攻击者通过显式操纵栈对象所造成的栈溢出错误，通过引用访问堆对象则有助于实现不同的垃圾回收方式：1. SIMPLE 默认采用基于类型作用域的自动垃圾回收，即，在类型退出作用域时，所有依赖于该类型的堆对象被系统自动检测并释放；2. 如果用户要求对堆对象进行显式释放，则 SIMPLE 将堆对象的控制工作完全交给用户，与此同时，保障程序安全性的责任也交由用户承担。这种内存管理方式迫使程序员意识到内存访问的灵活性与保障程序安全性之间的代价。

2.5 可扩展性

可扩展性（scalability）是指程序员可以通过较少的工作量对程序进行扩充和更新。SIMPLE 所提供的语言成分可以辅助程序员写出具有良好可扩展性的程序：

• 层次库结构。与 Ada 类似，SIMPLE 的层次库结构通过后扩库单元来实现，后扩库单元是指在父模块的基础上根据应用需求增加新的单元，它在逻辑上位于父模块规格说明的后面。通常，后扩库单元被设计成为后扩模块的形式，因为这样做可以使得新得到的模块进一步被扩展。

• 相同规格说明的不同实现。SIMPLE 允许一个规格说明和多个体相对应，即相同程序单元可以具有多种不同的实现，这不仅体现了程序单元在实现方式上的多态性，而且有助于程序扩展，因为在应用需求变更的情况下，往往需要对程序功能提供新的实现，而相同程序单元与多种实现的对应方式保障了新增加的体不对规格说明和其他的体产生任何震动。

2.6 效率

效率（efficiency）是指用某种语言编写的程序在编译或运行时对时间、空间等资源的有效利用率。SIMPLE 通过以下方式来提高程序效率：

• 使用分别编译来提高编译效率。分别编译是指将大程序分割成若干相对独立的模块，使得编译器对各个独立模块编译的效果与大程序的整体编译效果相当。如果一个模块在编码完成后不再被修改，那么这个模块只需要编译一次，当把它添加到整个系统中时，不需要每次都对它进行编译，从而降低编译开销，提高效率。

• 便于优化。SIMPLE 是一个平台无关的语言，即 SIMPLE 适用于各种操作系统和后端平台。为了满足这一需求，我们计划在 LLVM（平台无关的通用编译器框架）上实现 SIMPLE 的编译器。SIMPLE 的所有语言成分都可以转换成 LLVM 中间代码形式，由于 LLVM 自身提供了各种优秀的中间代码优化策略，因此我们可以使用 SIMPLE 写出高效的运行时代码。

3 典型语言设施

SIMPLE 借鉴了现存语言在设计方面的经验，其中 Ada、C++和 ML 对 SIMPLE 语言设施的设计具有较大影响。在此基础上，SIMPLE 提出了一些具有代表性的语言设施，旨在为新型语言设计提供新颖的思想。

3.1 程序单元类型

随着面向对象范型的广泛应用，类被越来越多的程序设计语言所采纳，它被当作功能强大的复杂类型来使用。在大多数面向对象语言中，类不仅是构造对象的模板，而且是具有良好模块化特征的程序单元。与面向对象中类的概念相似，ML 及其后代语言中的结构体（structure）也是构建程序的单元，同时，它可以用于声明新的结构体，并当作 functor 的参数[28]。

受面向对象语言中的类与 ML 及相关语言中的结构体的启发，我们进一步探索将程序单元作为类型处理的可能性。在 SIMPLE 中，我们将程序单元与类型进行充分的融合，允许将过程、函数、类、模块、异常当作类型处理，这些程序单元不仅保持了它们用于构建程序的原始功能，而且具备了作为类型的所有特性，它们可以用于定义对象。通过程序单元类型定义的对象具有和其他普通类型的对象相同的性质。图 1 说明了如何定义包含了各种图形的模块类型 Shapes。图 2 则给出了模块对象的使用示例。该程序段通过模块类型 Shapes 定义模块对象 shapes_obj，shapes_obj 具有与普通对象相似的属性，它可以被用作子程序 Process_Shapes 的参数。

```
type Shapes is module
public
  type Shape is class
    x_coord:float;
    y_coord:float;
    type Area is function() return float;
    type Distance is function() return float;
  end;
  type Circle is new Shape with
    radius:float;
  end;
  type Triangle is new Shape with
    side_1,side_2,side_3:float;
  end;
  type Process_Shape_Objects is procedure();
private
  circle_obj:Circle;
  triangle_obj:Triangle;
end;
```

图 1　模块示例——定义用于描述图形的模块 Shapes

```
type Process_Shapes is procedure(shapes_obj:Shapes);
procedure body Process_Shapes(shapes_obj:Shapes) is
begin
  shapes_obj. Process_Shape_Objects();
end;
```

图 2　将模块对象 shapes_obj 作为过程的参数

将程序单元当作类型处理具有以下潜在的优势：

• 程序单元类型可以辅助程序员构建结构复杂的程序。由于程序单元对象具有和相应程序单元类型一样的属性，因此在构建与已有程序单元内部成员相同的其他程序单元时，我们可以将现有的程序单元作为类型直接定义程序单元对象来构成其他程序单元的组件，而不需要通过编程的方式在其他程序单元中添加重复的代码，从而提高了编码效率和代码复用率。

• 程序单元类型有助于创建第一类对象(first class object)。在函数式程序设计中，函数允许被当作第一类对象(即函数像普通对象一样可以作为参数进行传递)，这种方式有助于提高编程效率。在 SIMPLE 中，所有程序单元都允许像函数式语言中的函数一样被当作第一类对象进行处理，这意味着过程、函数、类、模块、异常类型的对象都可以被当作子程序参数和函数返回值，图 2 中将模块类型 Shapes 作为函数形参类型就是利用程序单元类型创建第一类对象的典型示例。

• 程序单元类型使得多态性得到了更广泛的体现。在面向对象语言中，多态性往往体现于对象(这里特指类的实例)类型的动态绑定。而在 SIMPLE 中，不仅普通类的对象具有多态性，所有程序单元对象也可以像普通类的实例一样进行动态绑定。比如，假设 M 表示模块类型，那么类型 M 的引用可以指向任意 M 或它的派生类型的对象。这样，在对 M 的引用进行解引用时，就需要动态解析这个引用实际所指示的程序单元实例。

• 程序单元类型保障所有程序单元遵循语法相似性和语义一致性。在语法方面，将程序单元作为类型处理后，它们具有相似的使用方式，方便程序员使用类似的语法规则编写程序单元；在语义解释方面，程序单元类型具有普通类型的所有特征，是用于构造具有相同属性的其他程序单元的模板。

3.2　程序单元模板

诸多流行程序设计语言将泛型作为类型安全的多态容器，C++最先提出模板的概念来支持泛型程序设计，通过对 STL 库的不断扩展，C++的各类模板得到了广泛应用。随着泛型概念的普及，越来越多的语言开始支持泛型程序设计，例如，Java 5.0 版本开始提供泛型设施。但不同语言对泛型的支持力度依然存在差别，具体而言，像 Ada 和 C++一类语言支持将所有类型(包括基本数据类型和类类型)抽象成为泛型(模板)参数，这种泛型处理方式虽然使得泛型的应用范围更广，但代价是增加了检查泛型(模板)参数类型的难度，因为编译器无法通过静态检查保障模板形参的安全性，所以无法单独编译模板，只能在模板实例化时检查模板实参，对实例化单元进行单独编译。Java 和 C# 等语言则采取另外一种泛型处理方式，它们规定只有类或接口可以被泛化为泛型参数，并且在定义泛型单元时需要添加对泛型参数的限制，这样，编译器可以对泛型形参进行静态类型检查，直接编译泛型单元，从而减轻了泛型处理的难度，但也限制了模板参数的普及范围。

SIMPLE 的泛型设施主要借鉴 Ada 和 C++的处理方式，允许所有类型作为模板参数类型，并在此基础上对模板的应用范围作了进一步拓展。与大多数语言仅支持函数模板和类模板不同，SIMPLE 支持将各种程序单元(函数、过程、类、模块、异常)作为模板来处理，其中模块模板的作用类似 ML 语言中的 functor，用于创建具有相似属性的实例化模块单元。图 3 是一个模板示例，这个程序片段用来定义包含队列与栈两种数据类型的模块 Container。为了使所定义的数据类型支持对各种类型元素的操作，我们将 Container 定义成模块模板，它具有模板参数 T 和 S，用户可以通过将 T 和 S 绑定到具体类型实现模板实例化来定义不同元素类型的队列和栈。

```
type Container is(T,S)module
public
  type Queue is class
    type Push is procedure(item:in T);
    type Pop is function return T;
    type Is_Empty is function() return bool;
  end;
  type Stack is class
    type Push is procedure(item:in S);
    type Pop is function() return S;
    type Is_Empty is funtion() return bool;
  end;
private
  queue_obj:Queue;
  stack_obj:Stack;
end;
```

图 3　模板示例——定义容器模块模板 Container

SIMPLE 支持显式和隐式模板实例化。显式实例化表示在程序中显式提供实参类型与相应形参的绑定，得到一个模板的实例化类型，然后通过实例化的类型来声明对象；隐式实

例化则表示通过模板名与实参直接声明对象。与显式实例化相比，隐式实例化不会得到模板实例化类型，需要编译器自动进行类型绑定和安全性检查。图 4 给出这两种实例化方式的对比，隐式实例化无需给出实例化类型的具体定义，而显式实例化需要先将模板类型 Container 实例化为具体类型 Container_Int。

```
// 模板的隐式实例化
container_obj:(int,int)Container;
// 模板的显式实例化
type Container_Int is(T≥int,S≥int)Container;
container_obj:Container_Int;
```

图 4　模板实例化示例——隐式实例化和显式实例化

与 C++类似，SIMPLE 也支持模板特化和模板部分特化。模板特化表示将模板形参显式绑定为实参类型并为参数绑定后的模板提供一种特定的实现，模板特化体现了软件开发中逐步求精的思维方式。图 5(a)和图 5(b)分别是模板特化和模板部分特化的示例。在图 5(a)中，模板形参 T 和 S 均被关联为类型 int，程序员需要为实参均为 int 的模板 Container 提供一种独立的实现(即特化模板)，之后所有实参为 int 的模块对象都会由 Container 的特化模板而非普通模板进行实例化。在图 5(b)中，仅模板形参 T 被关联为类型 int，程序员需要为这个部分特化的模板提供独立于原模板的实现，之后在模板实例化时只需要将 S 绑定到具体类型便可以用这个部分特化模板来定义实例化类型。

```
type Container is(int,int)module;
```

(a)　模板特化示例

```
type Container is(int,S)module;
```

(b)　模板部分特化示例

图 5

3.3　规格说明与实现

Ada 是采用库规格说明与体相分离之机制的代表语言。在 Ada 中，库规格说明被当作接口，规格说明中所声明的成员为外界可见，库体则依赖于相应的规格说明并对外界隐藏规格说明的具体实现。这种机制的优点在于体的变化不会引起规格说明的震动，有效保障了规格说明的稳定性。对大型软件系统而言，大量的程序单元互相关联并彼此影响，但事实上，真正相关的代码仅仅是程序单元的规格说明而非整个程序，因此，规格说明与体相分离的机制有利于更清晰的组织代码，提升软件开发效率。Ada 之后所出现的诸多语言都采用了这种机制的衍生品——接口，比如，Modula 家族中的所有语言(Modula、Modula-2、Oberon、Object Oberon、Oberon-2、Modula-2+、Modula-3)都采用接口和模块来构建程序，它们有效分离功能说明和具体实现，使程序具有很高的模块化程度。

但是，Ada 规定每个规格说明只能与唯一的体相对应，这限制了模块功能实现方式的多样性，SIMPLE 则打破了这种约束，它允许一个规格说明与若干个体相对应，即一个程序单元可以具有多种不同的实现方式，这体现了程序单元在实现方式上的多态性。图 6 给出了使用这种特性的一个示例，该程序段中的 M_1 表示对模块 M 的另一种实现方式，它拥有和模板 M 相同的规格说明。从类型角度来看，类型 M_1 与类型 M 是等价的(即，M_1 可以被当作 M 的别名)，它们所声明的对象之间可以相互赋值，这与派生类型之间的关系是截然不同的。

```
type M is module
  type C_1 is class
    type Print is procedure();
  end C_1;
  type C_2 is class
    type Print is procedure();
  end C_2;
  type Print_Objects is procedure();
private
  c_1:C_1;
  c_2:C_2;
end M;
module body M is
  procedure body Print_Objects is
  begin
    c_1. Print();
    c_2. Print();
  end Print_Objects;
end M;
type M_1 is M;// M_1 是模块 M 提供了另一种实现
module body M_1 is
  procedure body Print_Objects is
  begin
    c_2. Print();
    c_1. Print();
  end Print_Objects;
end M_1;
```

图 6　一个规格说明对应多种实现的示例

与接口这一概念相比，SIMPLE 中单个规格说明与多种实现相对应的概念具有不同的特点：

• 这种概念可以被所有程序单元(函数、过程、类、模块、异常)所使用。与此相比，接口的使用范围相对狭小，因为在面向对象语言中，接口只能被用作类或方法的规格说明，在 ML 及相关语言中，接口被当作结构体的抽象。据所知，尚无语言能够将接口的概念用于所有程序单元的抽象表示，因此单个规格说明与多种实现相对应的概念具有更广的应用范围。

• SIMPLE 规定每个规格说明必须对应至少一种实现，接口则允许没有任何实现相对应。与接口相比，规格说明具有用户提供的一种默认实现，从而保证每个规格说明都可以被直接调用。

• 接口在实现时允许对接口进行扩展，即在实现中增加接口声明中未定义的成员，且新增成员可以被设为外界可见。然而，SIMPLE 中的所有程序单元体必须严格遵照相应规格说明来实现，用户无法访问在体的内部所定义的成员，这种措施保障了程序单元的访问性规则和程序安全性。

3.4　内存管理

长久以来，内存管理都是程序设计语言的热点研究问题，不同语言针对内存管理采取了不同的处理方式。C/C++等语言将指针直接暴露给用户并将垃圾回收工作全权交由程序员处理，这种垃圾回收方式是极不安全的，往往造成严重的内存泄露或空指针现象。与此相反，自动垃圾回收的方式已经

被越来越多的语言所采纳，以 Java 为代表的一类通过虚拟机进行编译的语言几乎都采用了自动垃圾回收的措施。全自动垃圾回收虽然减轻了程序员控制内存空间的负担并保障了良好的内存安全性，但往往需要大量的时间开销，有时是许多实时系统所不能接受的。Objective-C 等语言则采用了半自动垃圾回收的策略，它为程序员提供了垃圾回收池，并根据程序员的需求随时清空垃圾，但是悬空指针以及内存泄露等内存问题依然没有从语言本身得到良好的解决。

虽然 SIMPLE 没有完全解决与垃圾回收相关的所有问题，但是提出显式区分堆对象与栈对象的概念为更好地进行垃圾回收提供了可行的探索渠道。与 Java 等语言将所有对象都存储于内存堆空间不同，SIMPLE 中的对象既可以位于栈空间，也可以被存储于堆空间，这依赖于程序员对对象的声明方式，所有通过关键字 new 声明的对象都是堆对象，它们具有独特的 ref 属性，即堆对象的引用标识。堆对象自创建时刻起就一直位于堆空间，直至被手动或自动回收。堆对象的手动回收完全依赖于用户，一旦程序员通过 release 语句对堆对象进行了操作，剩余的所有垃圾回收工作就全部交由用户处理。由于手动垃圾回收具有很大风险，因此 SIMPLE 不鼓励用户进行手动垃圾回收，系统会提供自动垃圾回收的功能。与 Java 虚拟机在某个时刻突发执行垃圾回收程序不同，SIMPLE 垃圾回收程序会在后台实时监视所有堆对象的状态，在程序执行期间，当某个类型的生命期结束（即无法访问到该类型）时，与这个类型相关的所有堆对象将被垃圾回收器自动回收。假设类型 T 是基类型 S 的派生类型，o 是类型为 T 的堆对象，那么 o 可以在以下两种情形下被引用：

· 通过派生类型 T 的引用 t_ptr 访问堆对象：

```
type T_REF is ref T;
t_ptr:T_REF:=o'ref;
```

· 通过基类型 S 的引用 s_ptr 访问堆对象：

```
type S_REF is ref S;
s_ptr:S_REF:=o'ref;
```

如上所示，堆对象既可以通过它自身类型的引用被访问，也可以通过基类型的引用被访问，因此堆对象的回收可以按照以下两步来进行：

1. 当类型 S 离开作用域时，所有类型为 S 的对象被回收；

2. 检查所有类型为 S 的引用，如果存在引用指向 S 派生类型的堆对象，则回收这个堆对象。

因为 SIMPLE 中不存在弱引用，所以这种自动垃圾回收策略保障了所有废弃的堆对象都可以被回收。与堆对象不同，栈对象始终位于栈空间，在生命期结束后被自动释放。栈对象不能像堆对象一样被引用，因此所有栈对象的赋值均采用按值传递的方式，这一规定杜绝了因为栈空间的释放所形成的空指针。

3.5 异常处理

不断增加的软件复杂性使得软件健壮性变得越来越重要，学者们提出了多种技术用于增强软件健壮性，其中异常处理机制是有效保障程序健壮性的重要手段之一。自从 Goodenough[29] 最早提出异常处理的概念以来，这方面的理论已经越来越成熟，现代高级语言也都将异常处理作为必须的语言设施之一。

针对语言中的异常处理设施，现存两种流行的观点：在大多数语言中，异常被当作控制流结构来设计，比如 Ada、C++、Java、Module-3、ML、OCaml、Python、Ruby 等语言均采用这种设计观点。然而，Eiffel、C#、Modula-2、CommonLisp 等语言则采纳了另一种异常处理的观点，在这些语言中，异常被视为表达和处理程序非正常、不可预测以及错误运行状态的独立语言设施[8]。

以上两种观点均存在优势和不足。在基于前一种观点所设计的语言中，异常用于为程序执行过程中的意外情况建模，并且异常处理程序是程序整体的一部分，换言之，异常被当作构成程序的成分之一。这种异常处理方式有利于获取清晰的程序控制流和数据流。然而，这种方法存在不可避免的缺陷，即异常处理程序与程序正文的紧耦合会使程序变得复杂和庞大。与此相反，在基于后一种观点所设计的语言中，异常处理程序独立于程序正文，以合约的形式而创建。在程序执行过程中，如果合约中设定的条件被违反，那么异常处理程序将被激活，这种异常处理方式有效隔离了程序正文和异常处理程序，使程序结构更加清晰，但是在程序正文中注入的合约或断言语句会破坏整个程序的控制结构和数据流。

为了在程序正文中对异常进行有效分离，同时保持程序结构和数据流的完整性，我们提出了异常处理过程的概念，并将它运用于 SIMPLE 异常的设计。SIMPLE 的异常处理机制主要基于前一种异常处理观点，但是与 Ada、C++、Java 等语言不同，SIMPLE 将异常作为普通程序单元对待，而不是一种特殊的语言设施。与模块类似，异常内部可以拥有一个可执行的体以及若干函数和过程，其中异常处理过程负责根据具体的实参进行相应的异常处理，它的调用方式与普通过程调用相同。在程序运行时，如果正文抛出异常，那么异常可执行体或异常处理过程将被触发执行，待异常处理完毕，返回正文继续执行。

在异常恢复方面，大多数语言采用了独断的处理方式。比如，Ada 和 C++等语言规定在异常处理完毕后控制流直接转移到抛出异常的程序单元尾部，如果程序没有显式提供 finally 语句，那么 Java 会采取相似的异常处理方式。但实际情况是，在异常处理完成后，异常点之后的语句往往可以继续正常执行，因此将控制流直接转移到程序单元尾部的方式显得过于草率。基于此，SIMPLE 提供通过调用异常处理过程的方式处理异常，在异常处理过程执行完毕后，控制流将返回过程调用点（即异常点），继续执行异常点后面的语句。异常处理过程用于将程序执行时的非正常状态转化到正常状态，它的参数是将程序异常状态传递到异常处理程序的载体，异常处理过程的执行完成意味着异常状态恢复成功，程序的控制流的完整性可以得到有效保护。

图 7 中的代码示例说明了如何通过异常处理过程进行异常处理。程序中的 Divisor_Zero_Exception 表示除零异常，它的内部定义了用于处理分母为零的异常处理过程 Handle_Divisor，Int_Division 则是包含了异常处理方式的函数。如果程序在执行过程中出现异常，那么异常处理过程 Handle_Devisor 将被出现异常的函数 Int_Division 直接调用。因为异常处理过程的调用方式与普通过程调用一致，所以在过程 Handle_Devisor 返回（即异常状态恢复完成）后，控制流将转移到 throw 语句之后的程序点继续执行。

```
// 异常规格说明
type Divisor_Zero_Exception is exception
  type Handle_Divisor is procedure(divisor:int);
end Divisor_Zero_Exception;
// 采用异常处理方式的函数实现
function body Int_Division(dividend:int,divisor:int) return double is
  if divisor ==0 then
  // 直接调用异常处理过程
  throws
    Divisor_Zero_Exception. Handle_Divisor(divisor);
  end if;
  return(divident / divisor);
end;
```

图 7　异常处理示例——使用异常处理过程进行异常处理

4　多范型特征

多范型语言融合了函数式、逻辑式、命令式、面向对象、并发程序设计等多种语言范型。由于没有一种单一的语言范型可以简单有效地解决程序设计中的所有问题，因此学者们投入了大量精力来研究融合多范型的技术[9]。20 世纪 80 到 90 年代曾涌现出一大批多范型语言，比如 Erlang、OCaml、CommonLisp、Lena[10]、Oz[11]、Falcon 等，这些语言的出现为后续多范型程序设计的研究打下了良好基础。近年来，很多语言将增加多范型设施作为发展的重要渠道，Ada 2005[12-14] 和 Ada 2012[15-17] 增加了接口等设施用于更好地支持面向对象程序设计，C# 通过增加 λ 表达式来支持函数式程序设计，新型的程序设计语言 Scala[18] 充分融合了面向对象和函数式程序设计的特征，成为目前具有代表性的多范型语言。

基于现有多范型语言的成功经验，SIMPLE 在设计时融合了过程式、面向对象以及函数式程序设计范型。我们希望通过 3 种范型的充分结合使 SIMPLE 为程序员提供多样化的思考和解决问题的方法，适用于广泛的应用场景，同时使语言内核具有良好的伸缩性，利于今后针对不同范型的功能扩展。

4.1　过程式程序设计

在过程式程序设计语言中，程序由根据特定顺序所建立的过程构成，它反映了人类自顶向下的思维方式。过程式程序设计关注问题的步骤，这些步骤由计算机按照语句从前往后的线性顺序（偶尔会遇到分支或循环）进行处理。过程式程序设计语言包括 C、C++、Fortran、Pascal 和 Basic。

SIMPLE 允许不通过类或模块，而由函数和过程直接构建程序，它为程序员提供了简单自由的编程方式。

4.2　面向对象程序设计

面向对象程序设计是过去 20 年最流行的程序设计范型。与其他范型相比，面向对象在对真实世界建模、提高程序的可靠性和可复用性以及减少软件维护代价等方面具有明显优势。事实证明面向对象程序设计在大规模软件开发领域可以发挥重要作用，创造巨大的商业价值。

抽象、继承、多态是面向对象语言所应具备的基本性质。SIMPLE 像传统的面向对象语言一样通过类支持以上性质，并将面向对象的概念运用在模块等程序单元之上。使用传统面向对象语言所写的程序往往会出现由于程序员对程序结构组织不当而产生过于庞大的类，从而影响程序的可读性和可维护性。为了使类保持简洁，SIMPLE 对类成员进行限制，只允许对象、过程、函数作为类的成员。SIMPLE 将类当作抽象数据类型与相关操作的集合，而将更复杂的功能交由模块进行处理。类和模块共同支撑了 SIMPLE 的面向对象特性，表 1 是 SIMPLE 中类和模块从不同角度上的对比。

表 1　SIMPLE 中类和模块的对比

	类	模块
粒度	小	大
功能	功能单一	功能丰富
内部成员	对象或子程序，不允许有嵌套类	任意语言设施，允许各种程序单元嵌套
实例化	类的对象	模块对象

4.2.1　抽象性

抽象性是面向对象程序设计的基本特征。在 SIMPLE 中，类和模块都体现了抽象的概念，其中类代表具有相同特征的对象的抽象，模块则是各种程序单元的封装体。

4.2.2　继承性

众所周知，继承性分为单继承和多继承两种方式，支持多继承的语言包括 C++、Common Lisp、Perl 以及 Python 等。由于多继承往往引起成员名冲突等问题，语言必须制定合理的规则来应对这些情况。Java、C# 等语言通过使用接口有效地规避了多继承带来的问题，Ruby、Perl、JavaScript、Scala 等动态语言则使用混入（mix-in）机制[19] 实现继承。

为了不引入更多新的概念以保障语言的简单性，SIMPLE 没有采用接口或混入机制，而是通过类型派生实现单继承。在 SIMPLE 中，包括类在内的所有程序单元（函数、过程、类、模块、异常）都可以作为基类型被派生，派生类型具有基类型的所有属性，是对基类型的扩展。SIMPLE 允许将派生类型对象显式转换为基类型对象，反之亦然。图 8 说明了如何定义类的派生类型，在该程序段中，Car 是基类，Small_Car 是派生类，派生类 Small_Car 在类 Car 的基础上增加了函数 Get_Max_Passengers，并对基类 Car 中所声明的过程 Start 和 Stop 进行重载。

```
// 定义基类 Car
type Car is class
public
  type Start is procedure();
  type Stop is procedure();
end Car;
// 定义派生类 Small_Car
type Small_Car is new Car with
public
  type Start is procedure();
  type Stop is procedure();
  type Get_Max_Passengers is function() return int;
end Small_Car;
```

图 8　基类和派生类示例

4.2.3　多态性

多态性表示一个实体可以以多种方式被实现，它在 SIMPLE 中以两种方式体现：

- 一个规格说明与多种实现相对应（详见第 3.3 节）。
- 堆对象的动态绑定。图 9 说明了子程序的动态绑定，

其中，Car_Ref 被定义为类型 Car 的引用类型，它可以指向所有由类型 Car 或 Car 的派生类型所声明的堆对象，c_2 是派生类型 Small_Car 的堆对象，car_ref 指向 c_2，在动态运行时刻被解引用。

```
type Car_Ref is ref Car;
c_2:new Small_Car;
car_ref:Car_Ref:=c_2'ref;
// 在动态运行时刻对 car_ref 进行解引用
car_ref'Value.start();
```

图 9　子程序动态绑定示例

4.3　函数式程序设计

函数式程序设计将计算当作数学函数的赋值，避免了程序中出现对象状态和可变数据，它关注函数的应用。纯函数式语言通常具备以下要素：第一类函数、高阶函数、纯函数、惰性赋值、模式匹配。因为其中一些概念有助于提高编写程序的灵活性，所以很多非函数式语言在新的语言版本中采纳了部分函数式语言设施。比如，C# 3.0 版本提供了 λ 表达式等函数式语言设施，Python 也增加了 lambda、filter、map、reduce 等内置函数[21]。

SIMPLE 从两方面支持函数式程序设计：1. 将函数作为第一类对象；2. 支持构建高阶函数和柯里化。这两种方式都有助于提高运用函数的灵活性。

4.3.1　第一类函数

第一类函数(first-class function)表示函数被当作第一类对象。在支持第一类函数的语言中，所有函数既可以被当作其他函数的参数、返回值进行传递，也可以将函数赋值给变量或存储于数据结构中。

在 SIMPLE 中，函数可以被当作类型，且函数对象具有和其他对象相似的性质，它们可以被当作第一类对象。不仅函数，其他所有程序单元对象(过程、类、模块、异常)都可以被当作第一类对象。这种处理方式不仅使不同的语言设施在语法规则和语义解释方面都具有一致性，而且拓展了第一类对象的应用范围。

4.3.2　高阶函数和柯里化

高阶函数既可以接收其他函数作为参数，也可以将函数作为返回值。第一类函数式是构造高阶函数的基础，在图 8 中，Integrate 是高阶函数，它可以接收函数 Fn 作为参数。

高阶函数允许柯里化，换言之，函数在作为参数进行传递时接收一个实参并返回接收下一个参数的新函数。SIMPLE 支持函数作为实参传递时携带一个或若干个实参。图 10 是使用高阶函数实现积分运算的示例，说明了如何在 SIMPLE 中实施柯里化，这个例子用于计算二重积分 $\int_0^1\int_0^1 xy\mathrm{d}x\mathrm{d}y$。函数 Integrate 有两种不同的声明，它是一个重载函数。在用于计算二重不定积分的 Integrate 函数体中，函数 *FnX* 被声明为局部函数，它用于计算在参数为 X 的情况下，积分式在 Y 维度上的积分。函数 *FnY* 体中所调用的函数 *Fn* 是一个典型的柯里化函数，它每次接收不同的 Y 值并返回一个带有参数 X 的表达式。积分式 $\int_0^1\int_0^1 xy\mathrm{d}x\mathrm{d}y$ 分别给定了 X 和 Y 两个维度的上下限 1 和 0，通过调用重载积分函数 Integrate，我们便可以算得二重积分的值。

```
// 声明函数 Integrate，用来计算不定积分
type Integrate is function(Fn:ref function(X:float) return float,Low:
float,High:float) return float;
function body Integrate(Fn:ref function(X:float) return float,Low:
float,High:float) return float is
begin
  return Fn(High)-Fn(Low);
end Integrate;
// 声明函数 Integrate 的重载，用来计算二重不定积分
type Integrate is function(Fn:ref function(X:float,Y:float) return
float,LowX:float,HighX:float,LowY:float,HighY:float) return
float;
// 与上述重载函数规格说明相对应的函数体
function body Integrate(Fn:ref function(X:float,Y:float) return
float,LowX:float,HighX:float,LowY:float,HighY:float) return
float is
  type FnX is function(X:float) return float;
  function body FnX(X:float) return float is
    type FnY is function(Y:float) return float;
    function body FnY(Y:float) return float is
      return Fn(X,Y);// 对于给定的 X 和 Y 值，计算积分结果。其
      中，Fn 是一个柯里化函数
    end FnY;
  begin
    return Integrate(FnY,LowY,HighY);// 计算 Y 维度上的积分值
  end FnX;
begin
  return Integrate(FnX,LowX,HighX);// 计算 X 维度上的积分值
end Integrate;
// 定义函数 F，返回定值积分结果
type F is function(X:float,Y:float) return float;
function body F(X:float,Y:float) return float is
begin
  return X * Y;
end F;
//利用上述积分函数来计算二重积分 ∫01∫01 xy dx dy 的值
Result:int:=Integrate(F,0.0,1.0,0.0,1.0);
```

图 10　高阶函数和柯里化示例

结束语　本文介绍了一种新型程序设计语言 SIMPLE，它将简单性、可读性、可靠性、安全性、可扩展性、效率作为设计目标，具有简明稳定的语言核心，支持过程式、面向对象以及函数式程序设计。该语言通过提出一些具有代表性的语言设施，弥补了部分现有语言中广泛存在的不足，为新型程序设计语言的探索提供了参考。SIMPLE 语言的 BNF 语法描述、语言手册等相关材料可参见 SIMPLE 主页 http://ise.nju.edu.cn/simple。

参 考 文 献

[1] Sebesta R W. Concepts of Programming Languages(Ninth Edition)[M]. Pearson Addison-Wesley, 2010

[2] Hoare C A R. Hints on Programming Language Design[C]// Proceedings ACM SIGACT/SIGPLAN Conference on Principles of Programming Languages. 1973

(下转第 44 页)

sing DEEM[J]. IEEE Transactions on Reliability, 2004, 53(4): 509-522
[32] 杜彦华，刘春煌，曹松. 行车安全综合监控系统的时序 Petri 网描述及验证[J]. 铁道学报，2005，27(4)：11-15
[33] 杜军威，徐中伟，王树梅. 联锁逻辑模型的安全性分析[J]. 计算机工程与应用，2007，43(2)：1-4，32
[34] 原菊梅，侯朝桢，王小艺，等. 考虑环境因素的分布式系统可靠性建模及其分析[J]. 控制与决策，2007，22(3)：309-312，317
[35] 张君一，谢里阳，王正. 基于广义随机 Petri 网的 CIMS 多任务可靠性研究[J]. 计算机工程与应用，2007，43(33)：26-28
[36] 张君一，谢里阳，王正. 基于 ESPN 的制造系统多任务可靠性研究[J]. 东北大学学报：自然科学版，2008，29(7)：1029-1032
[37] 庞善臣，蒋昌俊. 一种基于不变量结构分解的工作流性能分析方法[J]. 计算机学报，2010，33(5)：908-918
[38] 范贵生，虞慧群，陈丽琼，等. 基于 Petri 网的服务组合故障诊断与处理[J]. 软件学报，2010，21(2)：231-247
[39] 刘玲艳，吴晓平，田树新. 基于粗糙集和 Petri 网的随机流网络可靠性评价方法[J]. 控制与决策，2010，25(8)：1273-1277
[40] 刘玲艳，吴晓平. 基于灰色随机 Petri 网的构件化软件可靠性早期评估方法[J]. 海军工程大学学报，2011，23(1)：16-22
[41] http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html
[42] Kuo C-H, Huang Han-pang. Failure Modeling and Process Monitoring for Flexible Manufacturing Systems Using Colored Timed Petri Nets[J]. IEEE Transactions on Robetics and Automation, 2000, 16(3): 301-312
[43] Kuo C-H, Huang Han-pang. Failure Modeling and Process Monitoring for Flexible Manufacturing Systems Using Colored Timed Petri Nets[J]. IEEE Transactions on Robetics and Automation, 2000, 16(3): 301-312
[44] 陆文，徐峰，吕建. 一种开放环境下的软件可靠性评估方法[J]. 计算机学报，2010，33(3)：452-462
[45] 董云卫，王广仁，张凡，等. AADL 模型可靠性分析评估工具[J]. 软件学报，2011，22(6)：1252-1266

(上接第 8 页)

[3] Hoare C A R. The Emperor's Old Clothes[J]. Commun. ACM, 1983, 24(2): 75-83
[4] Pierce B C. Types and Programming Languages[M]. The MIT Press, 2002
[5] Wilson L B, Clark R G. Comparative Programming Languages (Third Edition)[M]. Addison-Wesley, 2001
[6] Jones R, Lins R D. Garbage collection: algorithms for automatic dynamic memory management[M]. John Wiley & Sons, 1996
[7] Goodenough J B. Exception handling: issues and a proposed notation[J]. Communications of the ACM, 1975, 18(12): 683-696
[8] Kiniry J R. Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application[J]. Exception Handling, LNCS, 2006, 4119: 288-300
[9] Budd T A, Justice T P, Pandey R E. General-Furpose Multiparadigm Programming Languages: An Enabling Technology for Constructing Complex Systems[C]// First IEEE International Conference on Engineering of Complex Computer Systems. 1995
[10] Budd T A. Multiparadigm Programming in Leda[M]. Addison-Wesley, 1994
[11] van Roy P. Multiparadigm Programming in Mozart/Oz [C]// Second International Conference MOZ. 2004
[12] WG9. ISO/IEC 8652: 2007(E) (Ed. 3). Ada Reference Manual [M]. 2007
[13] Barnes J. Ada 2005 Rationale: The Language -The Standard Libraries[M]. Springer, 2008
[14] Carlisle M. Automatic OO parser generation using visitors for Ada 2005[C]// Proceedings of the 2006 Annual ACM SIGAda International Conference on Ada. Albuquerque, New Mexico, USA, November 2006
[15] Schonberg S. Ada 2012 Intrim Report[C]// Proceedings of the 2010 Annual ACM SIGAda International Conference on Ada. 2010
[16] Barnes J. A Brief Introduction to Ada 2012[M]. The GNAT Pro Company, 2011
[17] WG9. ISO/IEC 8652: 2012(E). Ada Reference Manual[M]. December 2012
[18] Odersky M, et al. An Overview of the Scala Programming Language (Second Edition)[R]. Technical Report LAMP-REPORT-2006-001. 2006
[19] Matsumoto Y. Matsumoto Yukihiro code No Sekai[M]. Nikkei Business Publicaions, 2011
[20] Tate B A. Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages[M]. Pragmatic Bookshelf, 2010
[21] Beazley D M. Python Essential Reference (Fourth Edition)[M]. Addison-Wesley, 2011
[22] Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation[C]// Proceedings of the International Symposium on Code Generation and Optimization. 2004
[23] Scott M L. Programming Language Pragmatics (Third Edition) [M]. Morgan Kaufmann Publishers, 2008
[24] ISO/IEC DTR 19768[R]. Draft Technical Report on C++ Library Extensions. 2005
[25] ISO/IEC 14882: 2011. Information Technology-Programming Languages-C++[S]. 2011
[26] Sun Microsystems, Inc. Java™ Platform, Enterprise Edition (Java EE) Specification, v5[S]. 2006
[27] Microsoft Corporation. C# Language Specification Version 3.0 [S]. 2007
[28] Harper R. Programming in Standard ML (Draft: Version 1.2 of 11.02.11)[S]. 2011
[29] Goodenough J B. Exception handling: issues and a proposed notation[J]. Communications of the ACM CACM Homepage archive, 1975, 18(12): 683-696
[30] 徐宝文. 试论高级程序设计语言的设计与评价标准[J]. 南京航空学院学报，1987，19(2)