Ada语言的发展

吴 迪 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 Ada语言诞生于1979年,1980年被指定为美国军用标准,1983年被正式确立为ISO标准并投入使用。Ada所追求的主要目标是:程序的可靠性与可维护性、程序设计作为人的活动(强调程序可读性比可写性更重要)以及效率。Ada凭借其强大的功能、良好的可靠性以及对软件工程思想的优良体现在20世纪最后20年对程序设计语言的发展产生了重要影响。Ada广泛应用于高可靠、长生存期的大型软件研发,在军事、商业、公共交通、金融等领域的核心软件开发中发挥着重要作用。诸多欧美国家的国防与空中管制系统、交通运输系统、银行安全防卫系统等均使用Ada语言研制开发。迄今为止,国际标准组织先后确立过Ada 83,Ada 95,Ada 2005,Ada 2012等4个语言标准,新标准在旧标准的基础上均保持了良好的兼容性。从语言机制、应用、影响力等方面对Ada语言的发展进行全面的介绍和分析。

关键词 Ada,程序设计语言,强类型机制,程序包,分别编译,异常处理,类属单元,面向对象程序设计,并发程序设计,契约式程序设计,大型软件开发

中图法分类号 TP312 **文献标识码** A

Evolution of Ada Programming Language

WU Di 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 The Ada programming language, born in 1979, was designated as the USA military standard in 1980 and officially established as an ISO standard and put into use in 1983. Ada's original purposes are reliability, maintainability, readability and efficiency. Ada, with its robust features, good reliability and excellent software engineering ideas embodied, has exerted great influences upon the development of programming languages from 1980s to 1990s. Ada is widely applied to exploit high-integrated and long-lived large software and it plays dominant role in manufacturing key software in the areas such as military, commerce, public transportation, finance, etc. Many software systems, including systems of national defense and air control, transportation systems and bank security guarding systems, are exploited with Ada in Europe and America. In general, four standards (Ada 83, Ada 95, Ada 2005, Ada 2012) of the language are published as international standards by ISO in the past 30 years and each standard has kept good compatibility upon the former one. From the perspective of language mechanism, application and influences, a comprehensive introduction and analysis of Ada's evolution will be presented in the paper.

Keywords Ada, Programming languages, Strong typing mechanism, Program packages, Separate compilation, Exception handling, Generic units, Object oriented programming, Concurrent program design, Programming by contracts, Large software development

1 引言

Ada是20世纪70年代末美国国防部为了解决软件危机而研制的通用程序设计语言,它以Pascal语言为基础开发实现。Ada既继承了众多优秀高级语言的先进思想与设施,同时融合了C++等流行语言的功能,是至今涉及范围最广、耗资最多的语言设计工作的成果。

Ada不仅具有完善且可靠的类型系统、丰富的控制结构设施,而且提供了并发与实时处理、异常处理以及支持大型程序开发等方面的语言设施。同时,Ada广泛支持结构化与模块化程序设计,体现了数据抽象、信息隐藏和封装原理。此外,Ada通过规格说明与实现、逻辑设计与物理设计相分离的手段将不同性质的问题分别考虑。Ada所采用的许多优秀的软件工程和语言设计理念为之后出现的一系列高级语言的设计产生了重大影响。

鉴于Ada的上述特点,其适用于开发大型、复杂、需要进

到稿日期:2013-10-02 返修日期:2013-11-25 本文受国家自然科学基金项目(61170071)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;徐宝文(1961-),男,博士,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。

行长期使用与维护且对可靠性要求较高的软件项目，尤其适用于开发实时与嵌入式软件系统，如 C⁴ISR(指挥、控制、通信、计算机、情报及监视与侦察)等大型军事与国防系统、航空航天系统、通信系统、过程控制、监控系统等[25]。

Ada 所提供的具有可重用性的语言设施为提高软件可重用性做出了重要贡献，同时它在语言环境、方法学以及技术等方面进行了良好的标准化。在上世纪八九十年代，Ada 的广泛使用大幅度遏制了激增的软件开发费用。但是 Ada 程序支持环境过于依赖语言本身，不仅限制了 Ada 环境的应用范围，而且增加了高效率编译器等相关工具的开发费用，因此美国国防部最终放弃了对 Ada 研发的经济和政策支持，致使 Ada 走向衰落。

自 Ada 诞生至今的三十多年间，计算机硬件迅速发展，计算能力迅猛提升，为适应不同的计算模式和软件需求，程序设计语言的内部设施和语言环境都发生了巨大改变，Ada 的设计也随之不断扩充发展。纵观 Ada 83，Ada 95，Ada 2005，Ada 2012 这 4 个语言标准，我们发现 Ada 在语言机制(包括面向对象程序设计、并发程序设计、契约式程序设计、大规模程序开发等)方面作了广泛而深刻的改进。

本文首先介绍 Ada 标准的修订过程，其次针对 Ada 的基本语言设施进行说明，然后详细分析 Ada 语言设施的扩充和语言机制的改进，继而讨论 Ada 的成就与不足，之后阐述 Ada 的现状与展望，最后浅析 Ada 对其他语言设计的影响。

2 Ada 语言标准

2.1 源起

Ada 语言的历史最早可以追溯到 1970 年代。当时美国国防部(DoD)在其所属项目中使用了 450 多种纷杂而多样的程序设计语言，这使得他们在军用软件开发与维护中遇到了大量问题；与此同时，1960 年代中期所出现的软件危机造成的影响依然存在。为了提高军用软件的可维护性以及消除软件危机所带来的影响，美国国防部成立了高级语言工作组 HOLWG(High Order Language Work Group)来寻找或研制一种三军通用的高级程序设计语言。HOLWG 的工作分 3 个阶段进行：

HOLWG 的第一阶段工作是对通用语言制定一套规范，并以该规范中的要求对当时已存在的语言进行鉴定和审核。经过广泛讨论和逐步细致的推敲，HOLWG 先后起草了名为稻草人(Strawman，1975)、木头人(Woodman，1975)、锡人(Tinman，1976)、铁人(Ironman，1978)、钢人[1](Steelman，1979，以下简称钢人需求)等 5 个关于语言规范要求的文件，其中钢人需求的文本表述最为严谨且规格说明最为详细。经过严格审核，美国国防部最终决定将钢人需求作为通用的语言标准。在确定通用语言标准的同时，HOLWG 开始对当时存在的所有高级语言进行重新考察，但是没有发现完全符合钢人需求的任何语言。最终，美国国防部决定以 Pascal、Algol 68 或 PL/I 3 个语言之一为起点，设计一种满足钢人需求的通用高级程序设计语言。

在确立了通用的语言标准(即钢人需求)之后，美国国防部对新语言的研发给予了更高的重视，他们要求所设计的通用标准语言必须是高质量的。与此同时，美国国防部意识到如果所设计的语言被国防部门之外的社会完全接受，那么他们将获得巨大的经济效益和军事影响力。因此，他们决定采取全球招标的方式来征集语言设计方案。在招标过程中，美国国防部先后收到了来自国内外 17 家单位的投标并选取了其中的 4 种方案进入最后的角逐。1979 年，在向全球专家征集反馈意见后，美国国防部决定采纳来自法国 Honeywell 公司以 J. D. Ichbiah 为首的团队提出的语言设计方案，并取名为 Ada，以纪念世界上第一位程序员 Augusta Ada Lovelace 女士[8](Ada 生于 1815 年，是英国大诗人拜伦之女，她在计算机鼻祖 Charles Babbage 的影响下描述了分析机和差分机如何进行编程，最早给出了计算机程序设计中的变量、递归等诸多思想，被公认为第一位计算机程序员)。1979 年 6 月，Ada 的参考手册首次公布。这是 HOLWG 第二阶段的工作成果。

HOLWG 第三阶段的工作目标是精炼 Ada 的定义，通过使用 Ada 编写各类程序对这个新语言进行广泛的测试和评价。1980 年 4 月，在对 Ada 进行修订后，美国国防部公布了 Ada 语言参考手册的修订本并于当年 12 月将 Ada 语言批准为美国军用标准 MIL-STD-1815A，以此纪念 Augusta Ada Lovelace 女士。至此，HOLWG 为美国国防部寻找或研制三军通用的高级程序设计语言的工作圆满完成。

2.2 Ada 83

1983 年，在对语言设计进行少量修改后，ISO 组织正式将 Ada 语言确立为 ANSI 标准，这就是著名的 Ada 83 标准[4](以下简称 Ada 83)。

Ada 83 遵照钢人需求而设计，并接受了当时全球语言与软件学者的讨论，其在诞生之时被公认为功能最强、可靠性最高、最能体现软件工程思想的高级程序设计语言。其在问世后已在许多领域得到了广泛而成功的应用。

Ada 83 所追求的主要目标是：程序的可靠性和可维护性、程序设计作为人的活动(强调程序可读性比可写性更重要)以及效率。为了达到这些目标，Ada 83 提供了一系列完备的语言机制与功能强大的语言设施，如强类型、程序包、任务、分别编译、类属设施、异常处理等。本文第 3 节将对这些基本语言设施进行具体介绍。

2.3 Ada 95

为了在保持软件可靠性等优良特性的基础上进一步增强 Ada 语言的灵活性与可扩展性，美国国防部与美国国家标准协会(ANSI)于 1988 年决定对 Ada 作一次大的修改并为此成立了 DoD Ada 联合计划办公室(AJPO)。AJPO 将修订 Ada 的计划取名为 Ada 9X 计划[14]，并成立了 7 个小组分别负责该计划的有关工作，另外还有来自世界各地的许多个人与组织也参加了各小组产生的文件的评审工作。Ada 9X 计划的总目标是：修订标准既要反映当前基本需求，又要尽可能对 Ada 界减少副效应、增加正效应[25]。

Ada 9X 计划从 1988 年 10 月开始实施，分三阶段进行：第一阶段，制定对 Ada 的修订需求；第二阶段，实际修改 Ada 语言标准；第三阶段，由使用 Ada 83 向使用 Ada 9X 过渡。修订小组对 Ada 83 设计过程中的研究报告以及许多其它 Ada 评注等进行了仔细分析，并收集了 Ada 83 投入软件开发的 10 年时间里实现和使用 Ada 的经验，标识了在研制 Ada 83 时为简化语言实现与降低危险而加上的限制，新增了支持面向对象程序设计的语言设施。Ada 9X 的雏形于 1992 年左右形成，1995 年完成最终修订，并被批准为 ANSI、ISO 与 IEC

标准，这个语言标准被称作 Ada 95 标准[15]（以下简称 Ada 95）。

Ada 95 由核心语言与附件两部分组成，并在 Ada 83 所强调的 3 个主要目标的基础上，进一步增强了语言的灵活性和可扩充性，同时对 Ada 用于支持不同应用领域的标准程序库予以扩展。与 Ada 83 相比，Ada 95 所做的调整主要有 3 方面：

- 增加了面向对象的语言设施；
- 引入了支持并发程序设计的设施——保护对象；
- 扩展了支撑大型程序设计的语言设施。

其他细节调整包括对任务机制、类属参数等方面的修改和扩充等。Ada 95 最大限度地使当时已经存在的 Ada 83 软件与 Ada 95 系统相兼容，从而使原有 Ada 83 程序无需修改或只需少量修改就可在 Ada 95 环境中运行。

Ada 95 标准的确立是 Ada 语言发展过程中的重要里程碑，该标准为语言的设计理念做出了很多根本性的扩展，并有效推进了 Ada 在大型模块化软件开发中的应用。之后的 Ada 2005 与 Ada 2012 标准虽然针对诸多语言设施进行了调整，但就价值与对语言发展的影响来看，Ada 95 标准无疑是最突出的。

2.4 Ada 2005

1997 年之前，美国国防部一直将 Ada 作为军用和国防软件开发的指定语言。然而，随着以 Java 为代表的一系列新型面向对象程序设计语言的兴起，Ada 语言的优势逐渐减退。此外，在当时高效率的 Ada 编译器也比较少。因此，美国国防部在 1997 年完全放开了对高级程序设计语言的使用限制，对 Ada 语言研发所投入的人员和经费支持也随之减少。此后，Ada 语言的管理及新标准的制定完全交由 ISO 标准组织旗下的 WG9 委员会负责。Ada 2005 标准[57]（以下简称 Ada 2005）是 WG9 委员会针对 Ada 语言进行独立设计的第一个新标准。新标准可以看作 Ada 95 标准的增强版，Ada 2005 以 Ada 95 为基础，针对 Ada 95 在高可靠性、长生存期以及大型实时嵌入式系统开发中所出现的不足之处予以弥补，同时借鉴其他流行高级语言中新颖且实用的语言设施对 Ada 进行补充和扩展。

Ada 2005 遵循了高度兼容性和一致性的原则，保持了以往 Ada 语言标准的语法风格。与 Ada 95 相比较，Ada 2005 主要在以下方面进行了改进：

- 通过补充部分面向对象语言设施使得 Ada 完全具备了面向对象特征；
- 对匿名访问类型的使用方式进行扩充，有效增强了访问类型的表达能力；
- 扩展任务调度策略并新增同步接口等并发概念，使得并发单元的调度方式更加灵活，同时提高了并发程序设计的安全性；
- 调整各种程序单元结构及其可见性规则，重点针对层次库结构的访问控制机制进行了完善；
- 大幅度补充了语言标准库，重点新增了功能强大的容器库，为支撑标准化的软件开发提供了便利，也增强了程序的可维护性。

2.5 Ada 2012

众所周知，任何程序设计语言每隔 5 到 10 年就要进行一次重新审核并对其中的语言设施进行完善的评估，然后制定新的语言标准，Ada 也不例外。进入 2010 年之后，WG9 委员会对 Ada 语言进行新一轮的考察和评估，并决定推出新的 Ada 语言的标准 Ada 201X。2012 年 12 月，新标准最终确立并被命名为 Ada 2012 标准[74]（以下简称 Ada 2012）。

该标准继承了 Ada 2005 的语言风格，提高了语言本身对程序正确性的保障并增强了部分语言设施在使用上的灵活性。其中最为重要的扩充在于契约机制的引入，Ada 2012 将契约式编程用于子程序断言、类型不变式等方面，为保障软件模块化开发的可靠性提供了重要手段。除此之外，Ada 2012 所做的扩充包含以下方面：

- 对部分程序单元结构及其可见性规则进行调整，比如允许函数使用 out 模式的参数等；
- 对并发程序设计在多核系统上的应用提供了一系列完善的任务分派策略；
- 通过增加新型表达式（如函数表达式、条件表达式等），增强了 Ada 语言的表达能力及程序的可读性；
- 对标准容器库进行扩充，为使用 Ada 进行标准化程序设计提供了更加有力的支撑。

Ada 2012 是对 Ada 2005 的有效补充与扩展，使得 Ada 在原有特点的基础上进一步增强了对程序可靠性的保障。过去三十多年间，随着程序设计语言的发展，Ada 语言版本不断地改进升级，Ada 2012 是迄今最完整、涵盖内容最丰富的 Ada 标准。

3 基本语言设施

作为通用程序设计语言，Ada 在最初的 Ada 83 标准中提供了一系列丰富而功能强大的语言设施，包括强类型、程序包、任务单元、分别编译、类属设施、异常处理等，其中以程序包为代表的诸多设施被其后的高级语言所采纳。在 Ada 发展过程中，这些设施作为语言核心被完整地保留下来。就 Ada 自身而言，Ada 83 标准在语言发展过程中起着奠基作用。本节对 Ada 83 中的基本语言设施进行介绍。

3.1 强类型

Ada 83 的首要设计目标是保障程序的可靠性与可维护性。为了实现这一目标，Ada 采用强类型设施，即具有类型安全性（Typing Safety）的类型系统，这是 Ada 被公认的语言特征之一。所谓强类型指以下 3 方面[38]：

- 程序中的所有对象必须先声明后使用；
- 对象在声明时必须指明类型；
- 对象上的所有运算必须保持类型不变，且得到指定类型的结果。

Ada 中的类型由 4 部分构成：类型名、定义在类型上的操作、类型的取值范围以及类型关系管理规则（比如显式类型转换）。类型所刻画的性质均是静态的，由编译器在编译时进行检查。这种静态类型机制保证了程序中对象的类型在程序执行期间保持不变。

Ada 中的每个类型名代表唯一的类型，即每种类型都是与众不同的，这被称作类型等价性。Ada 的类型等价性规定，在同一表达式中，不同类型的值不可以混用。

此外，为了在保障类型安全性的同时提高表达式的灵活性，Ada 支持显式类型转换，但禁止用户自定义类型的隐式转

换，这是因为显式类型转换是安全的，而隐式类型转换存在引发运行错误的风险，在很大程度上降低了程序的可靠性。

3.2 程序包

Ada 在设计之初采纳了很多新的程序设计概念，程序包是其中最具特色的一个。程序包是指逻辑上相关的一组实体的封装体，这里的实体包括类型、对象、子程序、其他的程序包以及任务和保护单元。程序包由两部分组成：规格说明和体。其中规格说明指出了该程序包可以利用的资源，亦即这个程序包和外界的接口。规格说明的编写和编译要先于使用它的程序单元。程序包体则描述了如何实现规格说明中指定操作的细节。程序包规格说明与体互相独立，可以进行分别编译[58]。

Ada 的主要设计者 J. D. Ichbiah 曾这样描述他所设计的程序包："Ada 的程序包结构很有吸引力。这个概念——大概是 Ada 的主要特色——清晰地分离了可见部分（用户接口）与程序包体（实现部分）。"

纵观程序包的设计和使用，其主要特征可以被归纳为以下 3 方面[58]：

• 它是程序设施的封装体，体现了良好的信息隐藏思想。程序包封装了一组逻辑相关的实体，它可以向外输出数据类型、对象或者其他程序单元，具有抽象类型和抽象子程序的程序包还可以向外输出抽象类型和一组特征操作。

• 程序包对规格说明和体进行了分离，提高了模块化程序开发效率。由于规格说明给出了程序功能的说明而隐藏了具体的实现细节，因此在大型软件的开发过程中，设计与维护人员只需关注和理解规格说明即已足够，这样做无疑减少了理解程序的工作量。此外，规格说明与体分离后，用户根据需要对体进行重新编写时，不需要对规格说明进行改动，因为体的修改不会给规格说明带来任何震动。

• 程序包是构建程序库的核心设施，为大型模块化软件开发提供了基础支持。Ada 程序包是基本的库单元，它可以被其他程序单元直接引用，而且是建立单元层次库结构的基本单位。

3.3 并发单元

并发处理对于系统程序设计和嵌入式系统设计十分重要，是这两类设计中的核心问题。Ada 之前的大多数高级语言没有对并发程序设计提供直接的支持，常常需要借助于操作系统的一些系统设施来实现并发操作。Ada 83 则通过任务设施实现了面向分布式系统的并发处理，使得用户能在高级语言一级上描述系统的并发活动，从而避免在使用操作系统低级通信原语时所遇到的困难。

和程序包一样，任务也分为规格说明和体两部分。其中规格说明指示了任务可使用的资源，任务体则描述了如何使用资源的细节，即任务被引发时执行的语句序列和接受其他任务调用的接收语句。

Ada 83 采用会合机制进行任务的通信和同步。会合机制是一种可靠的通信机制，但是效率不高，而且将任务作为并发单元是一种面向控制的并发。为了克服以上缺陷，Ada 95 采用保护对象作为并发单元。与任务类似，每个保护对象由规格说明和体组成。规格说明用于向使用者提供规定的访问协议，而体则用于实现规格说明所规定的访问协议。保护对象是一种面向数据的同步机构，它既综合了条件临界区和管程的思想，又比任务具有更强的模块性和更高的效率，也更加适合并发程序设计。本文第 5 节将对 Ada 中的任务单元展开进一步的分析与探讨。

3.4 分别编译

根据 Ada 程序单元规格说明与单元体分离的原则，主程序的体使用内部单元提供的任何设施时，只依赖这些单元的规格说明，即只有修改规格说明才会对主程序产生影响，若仅仅修改程序单元体，则无需对规格说明进行重新编译。因此，一个 Ada 程序可以一次也可以分几次编译。一次编译的源程序正文可以由一个编译单位也可以由多个编译单位组成。程序单元的规格说明或程序体都可以当作编译单位。分别编译是程序设计的实际需要，它的目标是将庞大而复杂的程序分解成比较简单、调试方便、易于修改的部分。

Ada 分别编译机制的优点在于提高了程序的开发进度。对于大型软件系统而言，许多程序单元之间互相关联、互相影响。但是从编译角度来看，真正产生互相影响的并不是所有的代码，而是程序单元的规格说明。分别编译机制可以确保只有那些受到影响的程序单元需要被重新编译。这样，程序单元的独立性显著增强，且多人分工协作的开发效率也得到提高。

3.5 异常处理

异常处理设施为软件可靠性提供了有力保障，使得程序在执行过程中出现异常时不至于直接终止而造成整个系统的崩溃。在 Ada 诞生之前，有些高级语言虽然已经提出了异常处理的概念并提供了部分异常处理设施（比如 PL/I 语言可以检测并处理 23 类不用的异常），但在实际应用中依然无法满足程序员对异常处理的要求。Ada 在 PL/I 和 CLU 两种语言的基础上，针对它们的异常处理机制进行补充和完善，提供了一组小而全的内建异常，从而为高可靠性软件的开发奠定了基础。

在 Ada 中，每个异常都由一个标识符即异常名指明。异常既可以是用户定义的，也可以是系统内建的。就异常的引发方式而言，Ada 既允许程序员通过 raise 语句显式引发异常，也支持由系统隐式引发异常。随着异常的引发，程序的正常执行将被抛弃，控制发生转移，下一步所采取的动作依赖于发生异常时程序结构的性质：如果异常发生在任务或保护单元之中，则这个并发单元将被标记为终止状态，即使在异常恢复后该并发单元也不会继续执行；若异常发生在普通的程序单元中，则将触发异常处理程序的执行。如果抛出异常的程序单元不包含异常处理程序，则异常将被隐式地传播到调用这个程序单元的调用点，即 Ada 的异常传播是沿着控制路径而非静态继承关系进行回溯的。待异常处理完毕后，程序控制将不会隐式返回到抛出异常的程序单元中，而是继续执行异常处理子句之后的程序。这就导致程序控制立即返回到更高的一层。

在 Ada 设计过程中，异常处理体现了当时大多数设计者及参评者的意见。在上世纪八九十年代里，Ada 一直是唯一被广泛使用的包含异常处理的程序设计语言。虽然 Ada 的异常处理机制依然存在一些问题，比如异常可以传播到外部作用域从而与可见性规则相冲突，而且对并发单元的异常处理并不合理（因为一个任务抛出异常时没有其他的任务处理它，这个任务便已终止），但是作为最早引入异常处理设施的

语言之一，Ada 的异常处理机制依然影响了后续的诸多语言。

3.6 类属设施

软件开发是一项工程性活动，效率是衡量工程成功与否的重要指标，而软件重用是提高软件开发效率的主要手段，类属设施的使用则为支持软件重用提供了巨大的帮助。类属设施最早出现在 CLU 语言中，Ada 之前的部分高级语言虽然也提供了类属设施，但是整体上存在以下两个问题：1）重用粒度太小，有些语言只提供类属子程序，如此规模的程序单元无法适用于软件工程的需要；2）由于类型化和强类型的要求，类属子程序的参数仅限于传统的特定类型。Ada 所提供的类属设施则有效避免了类似的问题。Ada 83 提供类属子程序和类属程序包两种类属单元，它既可以通过类属参数扩展重用参数的种类，也可以通过程序包扩大重用的粒度。

在对 Ada 中的类属单元进行实例化时，用户通过类属实参取代形参便可获得类属单元的一个实例。Ada 的类属设施不仅使得实例可以在源语言一级共享代码，从而提高代码重用效率，而且方便用户在程序库中编制更为通用的程序模块，用户只要通过实例说明就能方便地得到实用的程序模块，从而提高程序的可靠性和开发速度。除此之外，多样化的形式类型定义有利于拓展类属设施的应用领域，并且有利于代码优化。

4 面向对象程序设计

面向对象程序设计思想在高级语言中的最早体现可追溯到 SIMULA 67，但面向对象概念的全面发展和应用则是在 Smalltalk 80 产生之后。众所周知，一种面向对象程序设计语言必须提供对 3 个核心语言特性的支持：封装性、继承性、多态性。就 Ada 而言，最初的 Ada 83 在支持继承性和多态性方面缺乏系统性，因此被看作为基于对象的程序设计语言。Ada 95 对面向对象程序设计范型进行了全面的支持，它通过类型扩展有效弥补了 Ada 83 对继承性支持的不足，同时通过增加类域类型和抽象子程序、抽象类型完整地支持了动态绑定和多态性。至此，Ada 95 已经成为一个真正的面向对象程序设计语言。随着面向对象程序设计范型的普及和发展，各种更加完善的面向对象语言设施也不断涌现。Ada 2005 所增加的接口概念使得 Ada 具备了支持多继承的能力，而重写关键字 overriding 的加入则有利于提高编写面向对象程序的正确性。

本节将从面向对象程序设计的 3 个主要特征分别阐述和分析 Ada 语言不断完善的面向对象特征。

4.1 封装性

封装是指将抽象的数据和行为相结合，形成有机的整体，其中的数据和操作都是这个封装体的成员。在 Ada 诞生之前，封装的概念已经或多或少地体现在各种高级语言之中，比如数据类型、子程序就是最简易的封装体。在此基础上，Ada 语言对封装性作了进一步支持和扩充。在 Ada 中，封装体将单元的规格说明表现为外部可见的接口，而将实现的细节对外部隐藏，从而有效提高了程序的可维护性。当开发人员设计一个模块化程序结构的时候，程序的每个成分都应该进行封装，每个设计单元的接口定义应该尽可能少地暴露其内部工作。程序包即是一种逻辑相关实体的封装体，是 Ada 最具特色的程序单元之一，它使得抽象与信息隐藏原理得到完美体现[38]。例如，需要说明一个能够容纳所有图形类型的封装体时，可以作出如下定义：

```
package Shapes is
   type Shape;
   procedure Handle_Shape(S:in out Shape);
private
   type Shape is record
      ...
   end record;
end;
package body Shapes is
   procedure Handle_Shape(S:in out Shape) is
      ...
   end Handle_Shape;
end;
```

在这个程序包中，我们声明了用于进行图形描述的类型 Shape 以及相应的操作。程序包 Shapes 对数据类型和操作进行了有效的封装：包规格说明的公共部分定义了数据类型和子程序，私有部分列出了类型的实现细节；包体则对外隐藏了子程序的具体实现。用户可以通过包规格说明访问到类型 Shape 以及子程序 Handle_Shape，却无法访问这些成分的实现细节，从而有效达到了信息隐藏的目的。

4.2 继承性

Ada 83 提供了许多支持继承性的语言设施，比如 with 子句、use 子句、子类型、派生类型、类属单元等，这些语言设施有助于代码复用和程序扩展。但是，Ada 83 对继承性的支持能力依然有限，它不允许派生类型在父类型的基础上增加新的成分，无法真正实现类型扩展，因此 Ada 83 在所使用的继承机制是静态的，这使得 Ada 83 在继承关系的支持上受到很大局限。

为了使语言具有更强的面向对象特征，Ada 95 引入了完整的类型扩展功能：派生类型可以在父类型的基础上增加新的数据类型成分，从而使 Ada 的类型派生成为一种动态机制。Ada 95 允许通过关键字 tagged 将记录或私有类型声明为标记类型，只有标记类型方可被继承。例如，我们希望在之前所定义的类型 Shape 的基础上扩展出用于表示圆形的类型 Circle 的定义时，可以用如下说明：

```
package Shapes is
   type Shape is tagged;
   procedure Handle_Shape(S:in out Shape);
   type Radius is Integer 0..Integer'Range;
   type Circle is new Shape with Radius;
      --派生类型 Circle 继承自父类型 Shape
   procedure Handle_Shape(C:in out Circle);
      -派生类型重载子程序 Handle_Shape
private
   type Shape is record
...
end record;
end;
```

其中，标记类型 Shape 是父类型，类型 Circle 是 Shape 的派生类型，它的规格说明中不仅包含了类型 Shape 中所有的数据成员，增加了新的附加分量 Radius，而且可以对子程序 Handle_Shape 进行重载。

Ada 95 类型扩展机制很好地支持了扩展式程序设计，从而可以通过继承、修改或添加成分与运算的手段来对现有类型进行求精与扩充，其主要目标是支持在不需重新编译或重新测试的情况下就可以重用现有可靠软件[26]。

继承性分为单继承和多继承两种。单继承是指派生类只能从一个父类中继承属性和操作，而多继承则允许派生类从若干个父类中继承属性和操作。这两种继承方式在功能上基本等价，但使用单继承有利于表达结构简单、层次清晰的继承关系，在可读性与可维护性上相对较好，Ada 95 便采用了单继承方式。但当继承关系较为复杂时，采用单继承方式构建的层次结构更加庞杂，而多继承则可以简洁而有效表达出复杂的继承关系，因此 C++等语言选择采用多继承机制。然而，每个采用多继承机制的语言都需要解决如下问题：1）当派生类的多个父类继承自同一个基类时，派生类中会出现若干个相同的基类副本，从而增加了存储空间；2）若派生类的多个父类中存在相同的方法，则编译器无法判定派生类继承的是哪个方法，从而造成二义性。为了克服多继承的固有缺陷，部分高级语言（如 Java、C#）引入了接口的概念，单继承和接口的结合使用不仅可以达到与多继承同等的对复杂继承关系的表达能力，而且避免了多继承中出现相同基类与方法副本的问题。

为了增强对复杂继承关系的表达能力，Ada 2005 在保留单继承机制的基础上新增了接口的概念。接口是无成分与具体操作的标志类型，由关键字 interface 标识，其中所定义的操作只能是抽象子程序或空过程。Ada 2005 规定，若派生类型继承自某个接口，则在该派生类型的体中必须实现这个接口所声明的所有的抽象子程序。在 Ada 2005 中，派生类型只能继承自唯一的父类型，但它实现的接口则可以有多个。例如，当要说明一个位于三维空间中的圆形类型 Circle 时，可以对描述图形维度的接口类型 Dimension 以及获取图形维度的函数 Get_Dim 作出如下定义：

```
package Shapes is
  type Shape is tagged;
  type Dimension is interface;
    --接口类型
  function Get_Dim(X:Dimension) return Integer is abstract;
  type Radius is Integer 0..Integer'Range;
  type Circle is new Shape and Dimension with Radius;
    --Circle 继承类型 Shape 并实现了接口 Dimension
  function Get_Dim(Circle:Dimension) return Integer;
    --派生类型 Circle 实现接口方法 Get_Dim
private
  type Shape is record
    ...
  end record;
end;
```

其中，派生类型 Circle 在继承父类型 Shape 的同时实现了接口 Dimension，它继承了父类型和接口中所声明的所有属性和操作，必须实现接口 Dimension 中的抽象函数 Get_Dim。

与普通类型类似，接口也允许被声明为约束类型（Limited Types，约束类型的对象不允许被拷贝或赋值）。但是与标志类型不同，接口的约束性不具有传递性，即如果一个接口是约束类型，那么继承自该接口的派生类型或接口既可以是约束类型，也可以是非约束类型。

接口有效地扩充了 Ada 语言的继承机制，使得 Ada 在保留原有层次清晰的单继承方式的基础上增强了对复杂继承关系的表达能力。在构建大型模块化软件时，这种单继承结合接口的继承机制更有利于保障软件的可维护性和可扩展性。

4.3 多态性与动态绑定

除了无法实现类型扩展外，Ada 83 的另一个缺陷在于对多态性支持能力的不足：Ada 83 仅支持静态重载，而无动态绑定的概念，其重载机制虽然为程序设计提供了一定的灵活性，但要求在编译时确定参数的类型，因此 Ada 83 为程序设计所提供的多态性是很有限的。

Ada 95 通过增加类域类型有效增强了 Ada 语言对多态性的支持。在 Ada95 中，每种标记类型 T 都具有相应的类域类型 T'Class，T'Class 由以 T 为根的派生类型树中的所有类型组成——T'Class 既可以指示类型 T 本身，也可以指示类型 T 的任一派生类型，且派生类型的对象可以隐式地转换为类型 T'Class 的对象。所有类域类型的对象都具有一个标记，此标记用于在运行时标识相应派生类型树中的各个派生类型，这就是 Ada 95 引入标记类型的目的。

类域类型的出现为 Ada 程序设计提供了更强的灵活性，因为子程序不仅可以静态确定参数类型，而且可以在程序运行时动态地分派接收不同类型的参数。

除了类域类型，Ada 95 还增加了抽象子程序与抽象类型，二者在声明时均以关键字 abstract 标识。抽象子程序是一种没有体的子程序，不能被直接调用，但是作为原语操作，它可以通过继承的方式实现。抽象类型则有效支持了类型派生，一个抽象类型的派生类型可以为抽象类型规格说明中的所有抽象子程序提供相应的子程序体。抽象子程序和抽象类型可以用于更好地构建抽象数据类型。同时，将抽象类型和类域程序设计结合使用可以有效提高程序的灵活性和简明性。比如，假如希望创建如图 1 所示的类型树并实现子程序动态绑定，可以编写一个能同时处理同一类域类型中不同类型的过程 Process 如下：

```
package Shapes is
  type Shape is abstract tagged;
  procedure Handle_Shape(S:in out Shape) is abstract;
  type Circle is new Shape;
  procedure Handle_Shape(C:in out Circle);
  type Rectangle is new Shape;
  procedure Handle_Shape(R:in out Rectangle);
  type Triangle is new Shape;
  procedure Handle_Shape(T:in out Triangle);
  type Square is new Rectangle;
  procedure Handle_Shape(S:in out Square);
  type Reference is access all Shape'Class;
    --类域类型的访问类型声明
procedure Process
    Next_Shape:Reference;
  begin
    Next_Shape := ...;
    Handle_Shape(Next_Shape.all);动态绑定
  end Process;
end;
```

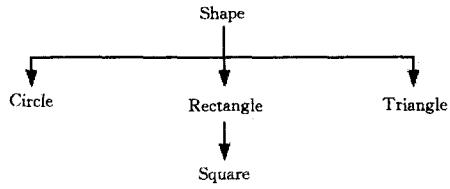

图 1　Shape 的类型图

其中，访问类型 Reference 用来对类型树（见图 1）中不同类型的对象进行指示。对过程 Process 而言，由于在编译时不能确定 Next_Shape 所指对象的具体类型，因此无法对 Handle_Shape(Next_Shape. all)所调用的子程序进行静态绑定。然而，因为 Reference 是类域类型 Shape'Class 的访问类型，所以 Next_Shape 的对象属性中包含一个用于指示该对象所属特定类型的标记，该标记用于在程序运行时刻确定 Handle_Shape(Next_Shape. all)调用的究竟是哪个派生类型所对应的重载子程序。

Ada 的主要设计目标之一是尽量保证程序的正确性，并且鼓励在编译时发现程序中的错误。从这个角度来看，虽然 Ada 95 通过引入类域类型扩展了多态性并且支持动态分派机制，但是它对子程序重写和重载的静态检查依然不够到位，从而造成在编写子程序时出现的一些不经意的错误被遗留到程序运行时刻[62]（比如程序员希望在派生类型中重写父类型中的一个子程序，但由于子程序名的拼写错误而引入了一个新的子程序）。Ada 2005 增加了关键字 overriding 来减少与子程序重写相关的错误的出现，从而有助于提高程序的可维护性。关键字 overriding 会提示编译器正在编译的子程序是对父类型中子程序的重写，若编译器发现父类型中不存在同名同参数的子程序，则会报错。为了兼容 Ada 95 版本，Ada 2005 将 overriding 用作可选关键字。

5　并发程序设计

并发程序设计是指由若干个可同时执行的模块组成程序的程序设计方法。并发程序设计语言通常包含一系列支持并发程序设计的设施以及用于控制并发对象间通信和同步的手段。

为了支持并发程序设计，Ada 83 最初将任务作为并发单元并使用会合机制来保障任务之间可靠通信。出于提高并发程序执行效率等目的，Ada 95 增加了保护对象的概念。Ada 2005 提出的同步接口等概念为并发程序设计提供了进一步支持，使得 Ada 语言更适用于实时并发系统的开发。Ada 2012 则侧重并发任务对多处理器的利用效率，通过新增同步栅等概念为并发程序单元在多处理器上的有效分配提供了更加完善和灵活的处理方式。

5.1　任务与会合机制

任务是能和其他程序单元进行并行操作的程序单元实体，是 Ada 的基本程序单元之一。从逻辑上讲，每个任务独占一个处理器执行；而在具体的实现上，多个任务既可以在单处理器上交替执行，也可以在多机系统和多处理器系统上执行[58]。

任务的定义类似程序包，包含规格说明与体两部分：任务规格说明定义了该任务对外界可见的入口，即其他任务可以利用的通信途径，每个入口包含一个隐式的入口队列，队列中的元素是在该入口上挂起的任务；任务体则定义了该任务或该任务类型对象的动作。需要说明的是，任务的规格说明中只允许入口的存在，而不允许子程序出现；同时，任务不允许包含私有数据类型，所有任务的共享数据对象须为全局对象或在任务体中显式声明。

一个并发系统同时存在多个并发执行的单元，为了完成一个给定的任务，这些并发单元之间通过通信和同步相互联系，因此实现并发模型不仅需要描述各并发单元，而且需要对并发任务间通信和同步有多种实现手段。Ada 83 将可靠性作为其主要目标之一，采用了具有优良可靠性的会合机制。会合机制采用一种完全同步的模型，即通信双方必须都到达预先确定的同步点，才开始通信。如果一方先到达同步点，则它必须等待另一方的到达。在会合机制中，通信双方是不对称的，一方是主动任务，通过入口调用发出请求，而另一方是被动任务，通过接受语句表示接受请求、准备会合。会合开始后，主动方等待服务方完成服务，返回结果，双方通信完成后再各行其事。

总之，完成一次基本的会合必须满足两个基本条件：1）需要外界对该任务入口的调用；2）需要该任务中有相应的接受。显然，这种"不见不散"式的会合机制是一种可靠的通信及同步方式。"生产者-消费者"问题是著名的并发问题之一，以下程序说明了使用 Ada 会合机制对该问题的解决方法：

```
task Producer is--生产者任务声明
  entry Start(How_Many:Natural);
end Producer;
task Consumer is--消费者任务声明
  entry Start(Break:Natural);
end Consumer;
task Buffer is--管程任务声明
  entry Insert(D:Natural);
  entry Take(D:out Natural);
end Buffer;
task body Producer is
  Local_How_Many:Natural;
begin
  accept Start(How_Many:Natural) do
    Local_How_Many :=How_Many;
  end Start;
  Consumer. Start(Local_How_Many +1);
      --触发消费者任务的执行
  for I in 1..(Local_How_Many +1) loop
    Buffer. Insert(I);
      --调用管程任务入口
  end loop;
end Producer;
task body Consumer is
  Over,Item,Result:Natural;
begin
  accept Start(Break:Natural) do
    Over :=Break;
  end Start;
  Result :=0;
  Buffer. Take(Item);
  while Item /= Over loop
  Result :=Result + Item;
```

```
  Buffer.Take(Item);
      --调用管程任务入口
  end loop;
  Print("Summe= " & Result);
end Consumer;
task body Buffer is
  Length:constant Natural :=10;
  B:array(0..Length-1) of Natural;
  In_Ptr,Out_Ptr:Natural :=0;
  Count:Natural :=0;
begin
  loop
    select
      when Count < Length =>
        accept Insert(D:Natural) do
          B(In_Ptr) := D;
        end Insert;
        In_Ptr :=(In_Ptr +1) mod Length;
        Count := Count +1;
      or
      when Count > 0 =>
        accept Take(D:out Natural) do
          D :=B(Out_Ptr);
        end Take;
        Out_Ptr := (Out_Ptr +1) mod Length;
        Count :=Count-1;
      or
        terminate;
      end select;
    end loop;
end Buffer;
```

其中,任务 Producer 和任务 Consumer 分别表示生产者和消费者,任务 Buffer 类似于管程,用于维护生产者和消费者所共享的数组。并发程序的执行通过语句 Consumer.Start;来触发,在并发程序执行过程中,生产者和消费者通过管程任务 Buffer 进行通信:生产者调用 Buffer.Insert 向共享数组中写入数据,而消费者调用 Buffer.Take 从共享数组中读出数据。作为管程任务,在同一时刻 Buffer 只允许唯一的任务对其入口进行调用,也就是说,生产者和消费者其中之一能对 Buffer 中的共享数据进行访问。

"生产者-消费者"示例充分展示了任务的实现与利用会合机制进行任务间的通信和同步。Consumer.Start;,Buffer.Insert;以及 Buffer.Take;这 3 条语句充当了任务之间的会合点,任务在这些程序点上发生会合,共同合作完成一些事务,然后继续各行其事。

5.2 保护对象机制

Ada 83 所使用的会合机制是一种高级的通信机制,利用它可以避免因使用诸如信号灯等低级通信原语而带来的困难,会合机制还为任务间的通信提供了良好的可靠性保障。会合机制虽然拥有上述优点,但是它依然存在一些问题:当有多个任务需要访问同一共享数据时,需要增加类似于管程的辅助任务对共享数据进行管理,这会大大降低系统的性能;而且,在某些情况下,如果因不小心而颠倒了有关抽象操作的次序,那么还会引起难以应付的竞争条件;另外,会合机制是一种面向控制的通信与同步机制,而对共享数据访问的保护与互斥则需要面向数据的通信与同步机制,故会合机制与流行的且得到公认的面向对象的开发方法不相一致[28]。

为了解决这些问题,Ada 95 引入了保护对象的概念。保护对象用于封装有关类型的数据对象并在不增加辅助任务的前提下实施对这类数据对象的同步访问。与任务类似,每个保护对象由规格说明及其对应的体组成。规格说明用于向使用者提供规定的访问协议,而体则用于实现规格说明所规定的访问协议。保护对象与任务的区别在于:首先,任务的可见部分只能说明入口,而保护说明的可见部分还能说明子程序;其次,在私有部分,任务只能说明入口,而保护对象则可以声明各种数据对象,这些数据对象就是所要保护的共享数据[38]。保护对象的使用者只能通过公共部分所提供的入口与子程序来访问被保护的共享数据,这种入口与子程序被分别称为保护入口与保护子程序,两者合称为保护操作。在保护对象中,各个保护操作是互斥执行的,即在任意时刻最多只能有一个保护操作处于执行状态。

保护入口体类似于过程体,其功能也很类似于保护过程,但区别在于:每个入口体对应一个入口栅。每当执行保护入口调用时,只有入口栅中指定的条件被满足时才会执行相应的入口体,否则,调用任务就被挂起在该入口的等待队列中,直到相应条件满足时才会继续执行。保护对象的行为受到入口栅的控制:当一个入口被调用时,首先计算入口栅的值,若入口栅值为真,则可立即调用执行入口体,若入口栅值为假,则调用者就被挂起到该入口的队列中,犹如在调用任务入口时的等待队列一样[28]。在保护对象中可以通过 requeue 语句将一个正执行保护入口调用的任务重新排队到同一保护入口或另一个入口的等待队列中。下面以打印机的并发使用为例来说明保护对象的用法。在使用打印机时,可能有若干个使用者共同使用一台打印机,因此必须保证打印操作的互斥。具体实现如下:

```
protected Print_Event is
  entry Print;--保护入口
  procedure Use_Printer;--保护子程序
private
  entry Reset;--保护入口
  Idle:Boolean :=True;--共享数据
end Print_Event;
protected body Print_Event is
  entry Print when Idle is
  begin
    Idle :=False;
    Use_Printer();
    requeue Reset;
  end Print;
  procedure Use_Printer is
    ...
  end Use_Printer;
  entry Reset when True is
  begin
    Idl :=True;
  end Reset;
end Print_Event;
```

其中,共享数据 Idle 表示打印机状态,如果当前打印机处于空闲状态(Idle 的值为 True),则请求 Print 的打印任务顺

利执行，否则当前任务被挂起在保护入口 Print 的入口栅上。待当前打印任务执行完毕后，打印机被重新置为空闲状态，从而触发在入口 Print 上等待的任务进入执行状态。

将本例与上节中的“生产者-消费者”示例进行比较后，不难发现保护对象是一种面向数据的通信手段，它既像管程一样把共享数据与互斥操作封装在一起，又像条件临界区一样使用了入口栅设施。因此保护对象既具有管程和条件临界区的优点，也克服了两者的不足。与 Ada 83 的任务及会合机制相比，Ada 95 的保护对象机制具有更好的扩展性、模块性与兼容性，同时并发效率更高，更加易于表达。

5.3 同步接口与同步栅

实践证明，保护对象机制有效增强了 Ada 对并发程序设计的支持力度，并为大型并发程序的开发提供了高效率、可靠性的保障。在保留 Ada 95 并发程序设计机制的基础上，Ada 2005 通过增加同步接口进一步增强了并发程序的可靠性，Ada 2012 则通过引入同步栅的概念扩展了 Ada 对实时系统的并发管理方式。

为了更加方便地实现并发单元间的互斥，Ada 2005 将接口与并发程序设计进行巧妙的结合——通过使用接口来达到并发程序单元之间的同步。同步接口是指用关键字 synchronized 来声明的接口，这种接口只能在任务或保护对象中加以实现，它可以有效保护接口中的并发操作和并发单元之间的共享数据，有助于提高并发程序的安全性。下面通过一个例子具体说明同步接口的用途。假设已经编写了用于读写操作的接口 RW，这个接口包含写操作 Write 和读操作 Read。在并发程序执行过程中，如果一个任务在执行 Write 操作时被其他任务的 Read 操作中断，那么后者将只能读取前一任务的部分内容，因此这样的并发单元是不安全的。我们可以使用同步接口解决这一问题：

```
type Sync_RW is synchronized interface and RW;
protected type Prot_RW is new Sync_RW with
    --通过继承同步接口定义保护类型
  overriding procedure Write(X:in Item);
  overriding procedure Read(X:out Item);
private
  V:Item;
end;
protected body Prot_RW is
  procedure Write(X:in Item) is
  begin
    V :=X;
  end Write;
  procedure Read(X:out Item) is
  begin
    X :=V;
  end Read;
end Prot_RW;
```

其中，同步接口 Sync_RW 从接口 RW 继承而来，保护单元 Prot_RW 实现了同步接口 Sync_RW。这样，接口 RW 中的操作 Write 和 Read 便成为保护子程序，它们在并发程序执行期间彼此互斥，避免了在执行过程中被中断，从而保障了任务在并发执行过程中的安全性。

Ada 2012 针对同步接口进行了进一步扩展，即允许用户通过关键字 Synchronization 对同步接口的操作进行附加说明，比如使用 Synchronization=>By_Entry 来指示入口队列中的任务进行重列队[69]。

此外，为了对实时系统中的并发单元进行同步管理，Ada 2012 将并发程序单元的入口栅与同步机制进行结合，增加了同步栅[65]的概念。同步栅是指带有判别式的约束类型，它在作为保护子程序的参数类型使用时可以有效控制并发单元的同步释放。例如，同步栅 Synchronous_Barrier 可以按照如下方式进行定义和使用：

```
type Synchronous_Barrier(Release_Threshold:Barrier_Limit) is limited private;
procedure Wait_For_Release (The_Barrier:in out Synchronous_Barrier;Notified:out Boolean);
```

其中，Release_Threshold 表示一个判别式，它用于对同步栅 Synchronous_Barrier 设定任务的计数阈值，Synchronous_Barrier 则被用作子程序 Wait_For_Release 的参数类型。每当有任务调用过程 Wait_For_Release 时都会触发同步栅上的任务计数：若并发任务个数未达到阈值，则该任务被挂起，进入等待状态；待被挂起的并发任务数目达到阈值时，所有在栅外等待的任务将被集中释放。Ada 2012 同步栅的使用可以使任务的释放达到同步，从而满足了部分实时系统的并发需求。

同步接口和同步栅未对 Ada 并发程序设计的机制做出根本性调整，而是作为现有会合机制与保护对象机制的重要补充，使得 Ada 的并发机制更加完善和可靠。

5.4 任务调度策略

对支持并发程序设计的语言而言，语言内建的任务调度策略是保障并发单元合理执行的重要保障。Ada 83 没有将任务的调度方式作为语言标准的一部分，而 Ada 95 仅提供唯一的 FIFO_Within_Priorities 预定义任务调度策略，即基于任务优先级的 FIFO 调度方法。但是在诸多并发系统的实施过程中，这种单调的调度策略并不能满足实时任务调度的应用需求。为了实现并发程序中的任务调度算法的多样化，Ada 2005 新增了 Non_Preemptive_FIFO_Within_Priorities、Round_Robin_Within_Priorities、EDF_Across_Priorities 3 种新的任务调度策略[62]。

Non_Preemptive_FIFO_Within_Priorities 是基于任务优先级的非可剥夺 FIFO 策略。在这种任务分配方式下，所有具有相同优先级的任务按照先进先出的顺序依次执行，正在执行的任务不能被更高优先级的任务中断。一个任务在执行过程中，其他所有任务均处于挂起状态，只有当执行中的任务执行完毕或被 delay 语句延迟时，才允许其他任务开始执行。这种任务调度策略被广泛用于高可靠性的并发程序开发领域。

Round_Robin_Within_Priorities 是一种基于时间片的任务调度策略。这种任务调度方式借鉴了操作系统中基于时间片的进程调度原理：具有相同优先级的任务被分配指定大小的时间片，任务在执行过程中根据时间片间隔进行切换，从而保证了任务调度的公平性。类似的调度策略在早期的并发程序设计中较为常见，Ada 2005 将其引入是为了完善语言内置的任务调度方式。

EDF_Across_Priorities 任务调度策略忽略了不同任务的优先级，仅根据任务的执行时限进行调度。EDF 是指最早时

限优先(Earliest Deadline First)。在这种任务调度方式中,系统首先对每一个任务的执行时间加以评估,进而选择执行时间最短的任务执行。EDF 被证明是一种高效率的任务调度方法。

除了增加新的任务调度策略外,Ada 2005 中还加入了 Ravenscar Profile[34] 对多种任务设施的使用进行控制。Ravenscar Profile 规定了任务执行的一组模式,这组模式可以看作针对任务所使用的编用的集合,其中规定了任务调度策略、任务优先级使用方式、是否检测任务阻塞等。Ravenscar Profile 在 IRTAW(International Real-Time Ada Workshops)会议上提出,并因会议地点 Ravenscar 而得名,后经讨论被 WG9 组织接受,且最终被列入 Ada 2005 标准。Ada 2005 采纳 Ravenscar Profile 的目的在于以此标准建议程序员在程序开发时尽量限制对复杂任务设施的使用,从而使并发程序的执行结果变得易于预测。实践证明,Ravenscar Profile 对于规范并发程序设计起到了很大作用,因此 Ada 2012 保留了 Ravenscar Profile 并将其扩充到多处理系统的应用上。

除此之外,任务负载均衡、CPU 时间监控、保护对象的优先级动态分配等策略也被纳入 Ada 2005 标准[60]。

6 契约式程序设计

20 世纪 70 年代,C. A. R. Hoare 提出了著名的 Hoare 逻辑并由此将断言(Assertion)机制运用在程序设计语言中。断言是指在适当的程序点添加预测语句并保证程序运行时刻预测语句一定成立。断言机制的应用为保障程序正确性与提高程序可维护性提供了重要手段,诸多语言将之作为不可或缺的语言设施:程序设计语言 Euclid 为构建可验证的系统程序而设计,其既允许将断言以注释的形式交给验证器进行处理,也允许将断言写成布尔表达式交由编译器进行编译,Euclid 中的断言成为验证程序的重要手段。除了 Euclid,部分其他的早期语言也提供了断言机制,其中 ADL(Assertion Definition Language) 语言是一种专门基于断言机制而设计的规格说明语言,该语言用于通过断言对软件行为和程序接口进行严格的刻画,从而为保障软件质量提供优良的支持。此外,C,C++,Java(自 Java 1.4 开始支持断言机制)等高级语言在设计时也将断言机制加入其中,但这些语言对断言的支持较为简单——断言通常体现为布尔表达式,往往用于对程序的运行时状态进行判断。

随着断言机制在程序设计语言中的成功使用,基于断言而产生的契约式设计思想于 20 世纪 90 年代被提出。与断言相比,契约式设计的层次更高,概念也更加完善,可以看作一种软件设计方法,这种方法要求软件设计者为软件组件定义正式的、精确的并且可验证的接口,从而为传统的抽象数据类型又增加了前置条件、后置条件和不变式。

契约式编程是契约式设计在程序设计语言上的应用,不仅为程序正确性和可维护性提供了保障,同时也是减少大型软件开发成本的有效手段。程序设计语言 Eiffel 最先为契约式编程提供了完整的语言设施,Eiffel 以"契约"思想为核心,建立了整个错误处理思想体系,它将软件错误产生的本质归结于对"契约"的破坏。Eiffel 通过提供前置条件、后置条件、不变式、变式、状态接受条件和状态结果条件等要素为程序构建了完整的契约框架。实践证明:凭借其对契约式编程的成功运用,Eiffel 有效保障了大型软件的开发质量,因此被公认为高质量系统开发语言。

受到契约式编程思想的影响,Eiffel 之后所出现的很多语言,尤其是面向方面的程序设计语言(如 AspectJ 等)都将契约式编程作为重要的语言机制。就 Ada 而言,它的核心目标之一是保障软件的安全性,而使用契约式编程可以为开发高质量软件提供有效的支持,将契约机制引入正符合 Ada 的设计初衷。早在 20 世纪 80 年代,即 Ada 83 诞生之时,就曾有学者提出将契约式编程的思想运用于 Ada,他们开发出一种基于 Ada 的注释性语言 ANNA(Annotation of Ada)[11]。ANNA 通过扩充 Ada 语法并增加新的形式注解的语法规则来实现,其中很多形式注解的规则采用了契约式编程的思想,如断言、子程序前置及后置条件等。然而,因为 ANNA 中的契约式编程语句均是作为程序注释形式出现,所以从本质上来讲它并未真正将契约机制应用于 Ada 程序之中。20 世纪 90 年代,Praxis 公司主持设计了第一个正式将契约机制与 Ada 程序设计语言进行组合而产生的 SPARK 程序设计语言[54]。如今,SPARK 被应用在空中管制系统等高安全性要求的领域。鉴于 SPARK 语言的成功实践,WG9 委员会的专家经过多番讨论最终决定在 Ada 2012 中正式加入契约机制。

Ada 2012 中的契约由前置条件、后置条件及不变式等概念组成,可用于类型检查、类型预测以及子程序维护等方面。另外,Ada 2012 还允许在程序编用中加入断言对程序的执行状态进行动态检测。

6.1 类型不变式与子类型预言

契约机制在类型系统上的应用包括类型不变式和子类型预言。类型不变式描述了程序执行过程中某一类型的对象所必须满足的属性,用于确保记录类型的内部成分之间的关系得以保持。我们依然以程序包 Queues 为例来说明类型不变式的作用。假如希望确保所定义的队列类型 Queue 中不存在相同元素,那么可以对类型 Queue 作如下定义:

```
package Queues is
  type Queue is private
    with Type_Invariant => Is_Unduplicated(Queue);
      --带有不变式约束的类型声明
  function Is_Empty(Q:Queue) return Boolean;
  function Is_Full(Q:Queue) return Boolean;
  function Is_Unduplicated(Q:Queue) return Boolean;
  procedure GetFront(Q:in out Queue;X:in Item);
  procedure DeQueue(Q:in out Queue;X:out Item);
private
  ...
end Queues;
```

其中,Type_Invariant 表示类型 Queue 所对应的不变式,它通过函数 Is_Unduplicated 保障了类型 Queue 中不出现相同元素。

对任意类型 T 的类型不变式而言,Ada 2012 规定必须在以下 3 个时刻进行类型检查:

- 类型 T 的对象被默认初始化时;
- 将其他类型的对象显式转换为类型 T 时;
- 部分子程序调用返回时,这类子程序包括:返回类型为 T 的函数,以及参数类型为 T 且传参模式为 out 或 in out 的过程。

除了类型不变式之外，Ada 2012 的契约式编程还体现于子类型预言(Subtype Predict)：程序员可以通过使用保留字 Static_Predicate 或 Dynamic_Predicate 来显式规定子程序的静态或动态属性。众所周知，1 年由 12 个月组成，每年 12 月到次年 2 月为冬季，那么我们可以使用子类型预言对子类型 Winter 作如下定义：

```
type Month is (Jan,Feb,Mar,Apr,May,…,Nov,Dec);
subtype Winter is Month
  with Static_Predicate => Winter in Dec | Jan | Feb;
```

其中，子类型 Winter 使用静态预言的方式说明了枚举类型的取值范围。类似于枚举类型的子类型完全可以通过静态方式加以约束，但是对其他很多类型而言，静态预言无法完成对其子类型取值范围的约束，因此必须使用动态预言来定义这些类型的子类型。偶数是指所有可以被 2 整除的整数，那么可以通过以下方式定义偶数类型：

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
```

由于模运算只能发生在运行时刻，因此一个偶数类型的对象是否满足约束条件只能在动态执行时刻确定。

6.2 前置条件与后置条件

为子程序添加前置条件与后置条件是契约式编程中常见的技术。前置条件是指在子程序声明中增加部分条件说明语句，只有满足这些条件才允许调用子程序；与此类似，后置条件是指在子程序声明中所添加的后置说明，这些条件必须在子程序调用完成时被满足。在程序执行时，子程序调用必须保证所有前置条件与后置条件成立，否则将抛出异常。Ada 2005 为子程序契约式编程提供了前置条件与后置条件这种语言设施。以程序包 Queues 为例，如果希望在执行元素添加操作 DeQueue 前确保队列为非满状态且在执行该操作后确保队列非空，那么可以在声明这个操作时为之添加前置条件和后置条件，针对过程 GetFront 的处理方法与之类似。具体的程序说明如下：

```
package Queues is
  type Queue is private;
  function Is_Empty(Q:Queue) return Boolean;
  function Is_Full(Q:Queue) return Boolean;
  --具有前置与后置条件的子程序声明
  procedure DeQueue(Q:in out Queue;X:in Item)
    with Pre => not Is_Full(Q),
      Post => not Is_Empty(Q);
  procedure GetFront(Q:in out Queue;X:out Item)
    with Pre => not Is_Empty(Q),
      Post => not Is_Full(Q);
  Queue_Error:exception;
private
    …
end Queues;
```

为了与 Ada 2005 版本相兼容，Ada 2012 允许通过添加程序编用的方式将程序中的契约置为不可见。若用户在程序中使用编用 pragma Assertion_Policy(Check)，则契约将在编译时生效；反之，若编用为 pragma Assertion_Policy(Ignore)，则程序中的契约将不会被编译。

7 大型软件开发

Ada 为支持大型软件开发提供了诸多语言设施，这些设施有助于对软件系统复杂性进行有效管理。Ada 83 中的程序包、类属单元、分别编译等语言设施为软件模块化开发提供了主要手段，Ada 95 中所加入的层次库机制为大型软件的开发提供了更好的控制、分解和扩展功能，Ada 2005 在层次库结构的访问控制规则上所作的调整则为大型软件的可靠性提供了更有力的保障，同时为程序的编写带来了更多灵活性。本节着重介绍和分析 Ada 层次库结构设计的变化对大型软件开发的支持。

7.1 层次库结构

在 Ada 83 中，规格说明与体相分离有助于我们在程序单元规格说明未发生改变的情况下单独编译相应的体或使用了这个程序单元的其他程序单元；类属设施可以辅助构造功能相似的程序单元；私有单元可用于分离接口与实现。这些基础语言设施对于支持大型软件开发无疑是很有效的，但 Ada 83 在用于开发某些大型系统时仍然存在比较突出的两个问题：1)它对私有类型的可见性控制比较粗糙；2)当要扩充现有系统以增加新的功能与设施时，要对各分别编译的程序单元作强类型检查，哪怕是很小的修改也可能会导致要对许多编译单元作大量的重编译与重链接工作[27]。

为了解决 Ada 83 在大型软件开发中所面临的问题，Ada 95 不仅提出了层次库的概念并提供了相应的设施，而且对 Ada 83 的分别编译设施与可见性规则进行了扩充，这些扩充既可以减少重编译，又有利于把一个大系统分解成若干个子系统进行开发，进而进行模块化的程序设计。

Ada 95 的层次库概念通过后扩库单元(Child Library Unit)来实现。后扩库单元可以是一个程序包或子程序。通常，后扩库单元被设计成为后扩程序包的形式，因为这样做可以使得新得到的程序包进一步被扩展。后扩库单元具有以下特点[27]：

- 在逻辑上隶属于其父库单元，犹如嵌入在父库单元的私有部分之前。对后扩库单元而言，父库单元的公共部分与私有部分都是可见的；
- 可以像嵌套程序单元一样用扩展名来命名，即在后扩库单元名字前加上父库单元名字前缀；
- 可以重复嵌套，即如果一个后扩库单元是程序包，那么它本身也可以有一个或多个后扩库单元。这样，后扩库单元可以被用作构造以父库单元为根的树形层次结构。

一个简单的队列类型应该包含入队和出队操作。因此可以如下定义队列程序包规格说明：

```
package Queues is
  type Queue is private;
  procedure GetFront(Q:in out Queue;X:in Item);
  procedure DeQueue(Q:in out Queue;X:out Item);
private
  type Queue is record
    …
  end record;
end Queues;
```

如果希望在不破坏原有规格说明的基础上对队列程序包进行扩充，增添判断队列是否为空、是否为满、是否存在相同元素等操作，那么可以使用 Ada 95 所提供的层次库机制将所要添加的部分放在后扩库单元中：

```
package Queues. Ev_1 is
```

```
  function Is_Empty(Q:Queue) return Boolean;
  function Is_Full(Q:Queue) return Boolean;
  function Is_Unduplicated(Q:Queue) return Boolean;
private
   ...
end Queues. Ev_1;
```

其中，Queues. Ev_1 被称为公共后扩库单元，这种后扩库单元对外界可见，用户可以通过 with 子句对公共后扩库单元进行直接访问。如果用户希望后扩库单元对外界不可见，那么可将其声明为私有类型。私有后扩库单元仅供其父单元使用，与公共后扩库单元在语法上的区别在于其由关键字 private 进行标识。一般而言，私有后扩库单元可见性规则与公共后扩库单元类似，但有两点例外：1)私有后扩库单元只能被以其父单元为根的分层子树中的各后扩库单元访问，而不能被该子树中的公共后扩库单元规格说明部分访问；2)私有后扩库单元的可见部分可以访问其父单元的私有部分。

层次库结构为大型软件的开发提供了很好的控制与分解功能。当一个大型系统无法用单个程序单元实现时，后扩库单元便可用来将较大的系统分解组织成若干子系统，通过后扩库单元实现的库单元的层次组织可用于对已完成的软件系统作特定扩充，以便使系统移植到具有不同配置的机器上，从而提高系统的可移植性[27]。

7.2 层次库结构的访问控制

在 Ada 中，有效运用程序结构是控制可见性与信息隐藏的重要手段。Ada 95 中层次库机制的加入更为大型软件的模块化开发提供了有效的辅助作用，同时增强了软件的可扩展性和未来对软件进行更新的灵活性。实践证明，层次库机制可以为大型软件的开发提供良好的支持。

Ada 95 的访问控制规则规定：父库单元的私有部分只对公共后扩库单元的私有部分以及私有后扩库单元可见，而私有后扩库单元对除父库单元之外的其他所有公共库单元均为不可见。这种访问控制规则的优点在于：1)可以有效地支持私有后扩库单元中的信息隐藏；2)可以保护父库单元私有部分的信息。然而，这种规则同样存在弊端：由于位于层次库结构底层的公共库单元不能访问上层的私有库单元，因此在复杂的层次库结构中，只有将大量信息存储于公共根节点的私有部分，才能保证底层库单元访问到需要的信息，由此便造成库结构顶层的公共库单元过于庞大，从而增加了软件维护的难度并降低了程序的易读性。

为了在不破坏 Ada 95 访问控制规则的基础上增强后扩库单元访问的灵活性，Ada 2005 增加了 private with 子句。private with 子句用于将私有后扩库单元的可见部分向公共后扩库单元的私有部分开放。下面通过对程序包 Queues 的扩展来说明 private with 子句的用途。首先定义 Queues 的私有后扩库单元 Queues. Ev_1：

```
package private Queues. Ev_1 is
  function Is_Empty(Q:Queue) return Boolean;
  function Is_Full(Q:Queue) return Boolean;
private
  function Is_Unduplicated(Q:Queue) return Boolean;
end Queues. Ev_1;
```

现在，我们希望为程序包 Queues 再定义一个后扩库单元 Queues. Ev_2，并在 Queues. Ev_2 私有部分中访问 Queues. Ev_1 公共部分定义的函数。依照 Ada 95 的可见性规则，Queues. Ev_2 无法直接访问 Queues. Ev_1 中的任何内容。因此，如果希望在 Queues. Ev_2 中访问函数 Is_Empty 和 Is_Full，那么必须在父程序包 Queues 的体中通过 with 子句对私有后扩单元 Queues. Ev_1 进行引用，并且在 Queues 的私有部分对函数 Is_Empty 和 Is_Full 进行声明，但是这种方式既复杂又使得父程序包过于臃肿。然而，Ada 2005 对类似情况的处理方式则更加简洁有效，它允许程序员利用 private with 子句在 Queues. Ev_2 的私有部分对 Queues. Ev_1 的公共成员进行直接访问：

```
private with Queues. Ev_1;
package public Queues. Ev_2 is
  function Is_Valid(Q:Queue) return Boolean;
private
  function body Is_Valid(Q:Queue) return Boolean is
  begin
    if Queues. Ev_1. Is_Empty(Q)
    or Queues. Ev_1. Is_Full(Q) then
      return false;
    else
      return true;
  end Is_Valid;
end Queues. Ev_2;
```

需要注意的是，private with 子句只允许公共库单元的私有部分访问其他私有库单元的公共部分，但私有库单元的私有部分对外界依然不可见，譬如本例中的 Queues. Ev_2 无法访问 Queues. Ev_1 在私有部分所定义的函数 Is_Unduplicated。private with 子句的出现不但没有对私有后扩库单元信息隐藏的作用产生任何影响，而且有效降低了层次库结构的复杂度，从而辅助用户实现类型明确、功能独立的库单元。

综上所述，Ada 95 通过提供层次库结构设施对 Ada 83 的分别编译机制和可见性规则进行了大幅度扩充。这些扩充不仅有效减小了程序重编译的代价，而且有利于大型软件系统进行层次化、模块化开发。Ada 2005 在 Ada 95 的基础上进一步补充了对层次库结构的访问控制机制，在不破坏大型软件层次库结构的前提下，有效降低了层次结构的复杂度，提高了大型软件的开发效率。

8 成就与不足

Ada 语言的研发自 1975 年从一系列程序设计语言需求开始，至 1979 年以钢人需求的确立而结束。在当时，美国国防部意识到，在资金密集型软件技术方面，虽然语言本身重要，但只是其中的一部分，所以又继续开发了称为“石人计划”[2]的语言环境需求以及称为“方法人计划”的软件方法学需求。因此，Ada 的成果不仅仅局限于语言本身，还涉及语言环境、方法学与技术。它们都以高度的可再用性来提高软件的可再用性。事实证明，Ada 将语言、环境和大型软件系统的方法学集为一体是一个很大的创新[7]。

就语言本身而言，Ada 所提供的一系列具有可重用性的语言设施为提升软件复用效率做出了重要贡献。其中，子程序、程序包、任务和保护对象等程序单元的规格说明为构造复合程序结构提供了功能明确的语法接口；强类型保障了模块形参和模块调用实参的一致性；规格说明与体分离使得信息

隐藏原理得到充分体现；类属单元为不同类属参数的模块提供了功能相似的实现，提高了代码重用效率；支持分别编译的程序库结构增强了软件模块的独立性和可扩展性，有利于提高软件开发进度。此外，比起 Ada 之前的程序设计语言，Ada 更好地支持了抽象机制。这个成就很突出，因为它增强了语言的表达能力。

Ada 良好的语言特性直接决定了它所带来的经济效益。在上世纪八九十年代，Ada 语言、环境、方法学以及技术的标准化在很大程度上遏制了激增的软件开发费用，同时提高了使用 Ada 所开发的工具的可再用性。

然而，在 Ada 带来巨大经济利益的背后也隐藏着一些缺陷：

1. 最初版本的 Ada 研发工作处于从顺序向分布式、从分时向交互式程序设计的过渡阶段。当时许多具有创新性的思想被加入到 Ada 的设计中，但由于 Ada 语言及其环境的设计尚未完结，而且设计 Ada 时所基于的软硬件均已过时，因此 Ada 83 的设计遗留下许多问题；

2. Ada 程序的分程序结构风格在 20 世纪 70 年代占据统治地位，但是进入 80 年代和 90 年代后，面向消息的分布式程序设计风格逐渐取代了分程序结构的地位；而且 Ada 中的模块化机制没有有机地结合起来；

3. 在软件生存期方面，Ada 基于 20 世纪 70 年代流行的“瀑布”模型而设计，它不能很好地支持原型开发等其他模型；

4. Ada 83 只反映了程序设计早期的分时技术的成果，无法体现交互式程序设计等新技术；

5. Ada 程序支持环境过于依赖 Ada 语言本身，不仅约束了 Ada 环境的应用范围，而且使工具开发变得更加昂贵。

Ada 95 的出现为解决上述问题提供了有效的手段，它不仅为支撑分布式、交互式程序开发增加了新的语言设施，而且具备了完整的面向对象特征，增强了对模块化程序设计的支持。然而，直至上世纪 90 年代末，Ada 程序支持环境对语言的依赖问题依然没有得到有效解决，Ada 编译器的研发工作耗费了大量的财力，但高效率的编译器却依然匮乏。除此之外，在 1987 至 1997 年间，美国国防部强制将 Ada 作为开发所有军用软件的标准语言，这引发了大多数软件工程师的强烈不满[66]。时至 1997 年，美国国防部最终放弃将 Ada 作为统一的软件开发语言，并同时大幅度削减了 Ada 研发工作的财政支出，从而导致 Ada 逐步走向衰落。

9 现状与展望

如今，Ada 虽然已经退出了主流的程序设计语言之列，但依然被广泛应用于需要高安全保障、高性能的大型软件开发领域。事实证明，对于需要大量开发人员、耗时若干年开发的复杂软件系统而言，使用 Ada 进行开发可以有效控制开发成本，同时有利于保障软件的可靠性与可维护性。在欧美国家，如今依然有组织和公司持续进行着 Ada 语言的研究和推广。作为 Ada 语言标准的制定者，WG9 组织不断对 Ada 的发展趋势进行评估和预测并制定适应 Ada 语言生存和发展的新标准；ARG 组织则负责为新的 Ada 标准添加评注以便用户更好地理解和使用 Ada，同时 ARG 还是 Ada 使用者与标准制定者之间的纽带，他们向 WG9 组织反馈 Ada 用户对语言改进的意见和建议。在学术界，诸多欧美学者致力于研究 Ada 的语言机制以及 Ada 在其他领域的应用，美国的 SIGAda(1998 年之前名为 TriAda)以及欧洲的 Ada-Europe 组织每年都会召开会议就与 Ada 相关的各方面议题展开讨论。在工业界，以 AdaCore 为代表的公司持续研发着以 Ada 语言作为核心的软件产品，其中 GNAT 集成开发环境具备相当优异的性能，它支持包括 Ada、Java、C++等多种程序设计语言的开发和编译，在稳定性和用户友好性上可以与其他任何商用集成开发和编译软件相媲美。

Ada 语言在对软件系统的可靠性具有严格要求的应用领域发挥着关键作用，这些领域包括军事、商业、公共交通、医药产业、银行和金融等。比如在著名的波音 777 客机的软件系统中，99%的代码由 Ada 语言编制而成；香港、伦敦、巴黎等城市的地铁控制系统的核心部分仍然运行着 Ada 程序；欧洲航天局(European Space Agency，ESA)与美国国家航空航天局(National Aeronautics and Space Administration，NASA)的诸多软件项目采用 Ada 语言进行研发。此外，AdaCore 公司曾于 2005 年推出一种以 Ada 为基础的子语言 SPARK，这种语言以强调程序正确性为核心，被广泛用于高可靠性软件系统的研发。英国军方目前正在使用 SPARK 语言开发下一代的空中交通控制系统 iFACTS[66](Interim Future Area Control Tools Support)。

我国是最早开展 Ada 语言研发工作的国家之一，在 Ada 计划一开始就有人投入对 Ada 的研究。自 20 世纪 70 年代末至本世纪初，曾有诸多国内学者对 Ada 语言成分、编译系统、软件开发环境等做过大量研究。我国也曾将 Ada 作为军用软件开发语言，修订并批准了《中华人民共和国国家军用标准 GJB 1383-92 程序设计语言 Ada》，该标准于 1992 年发布并于 1993 年开始实施。但随着美国国防部放弃对 Ada 语言在软件开发中的强制使用，Ada 的研究工作也随之进入低谷，最终使得 Ada 退出了主流程序设计语言的行列。从今天看，或许 Ada 很难再经历当年的辉煌，但经过对语言标准的逐步改进，Ada 在保留其最初可靠性与可维护性、可读性以及高开发效率等特点的基础上，结合了当代流行语言中优秀的语言特征来不断优化自身语言设计，使之更加适用于大型模块化软件的开发。我们认为未来的 Ada 语言将会继续在国防、公共交通、医疗、金融等领域的大型高可靠软件系统的研发过程中发挥核心作用。

10 影响力

Ada 所提供的语言设施(强类型、程序包、类属单元、规格说明与体分离、分别编译等)体现了丰富的软件工程原理，被其他语言广泛地借鉴和吸收。当今流行的面向对象语言在设计时或多或少都受到了 Ada 的影响。

以 Java 为例，它在设计实现时舍弃了 C++中诸多过于繁琐的语言机制，借鉴了一部分 Ada 的设计思路。与 Ada 类似，Java 同样采用强类型系统，这为保障语言的安全和健壮性提供了有力保障。程序包是 Ada 中最具特色的语言设施之一，是体现封装性和信息隐藏原理的典型代表。鉴于程序包在模块化程序设计中的重要作用，Java 也将之作为一种语言构造。另外，Java 中接口的概念用于将方法说明与实现相分离，这与 Ada 中单元规格说明与体分离的设计思想相一致，二者都是提高程序可读性的有效手段。

C++是在C语言基础上通过增加对象模型所形成的面向对象语言。虽然在很多方面C++与Ada有广泛的差别，但C++设计者在更新语言标准时从Ada的设计中汲取了不少经验。早期的C++标准中没有异常处理设施，这在保障软件可靠性方面是很大的漏洞，ANSI C++标准化委员会于1990年接受了C++的异常处理，此后，C++的异常处理被C++的许多种实现所采纳。它的设计基于Ada、ML等语言的异常处理机制，都包含由用户定义或库定义的异常。与Ada相比，C++的异常处理更加简洁，它不提供预定义的异常。此外，Ada使用的是单独异常处理，而C++使用的是随文异常处理。

Modula系列语言（Modula、Modula-2、Oberon、Object Oberon、Oberon-2、Modula-2+、Modula-3）在发展过程中也汲取了Ada的设计思想。与Ada相比，Modula所提供的语言设施更加精简。Modula语言的主要特点在于其强大的模块化机制，程序中的所有单元都是接口或模块，Modula中接口的概念对应Ada中程序单元的规格说明，而模块则对应了单元体。Modula语言不仅使用了类似Ada的强类型机制，其对数据类型的检测同样严格，而且增加了效仿Ada而设计的类属设施。此外，Modula-2+所加入的异常处理机制及Modula-3新加入的面向对象机制也都参考了Ada中相应语言设施的设计。

Ada虽然不是第一个使用类属设施的高级语言，但却有效解决了之前语言类属设施的重用粒度太小、类属子程序参数受限等诸多问题，为其后语言对类属设施的设计起到了巨大使用。随着程序设计语言的发展，为支持软件重用起到重要辅助作用的类属设施出现在越来越多的语言设计中：C++中的函数模板可以实现与Ada类属子程序相同的功能，同时二者对子程序的绑定调用都是静态的；Java 5.0中也加入了对泛型的支持；C# 2005中的泛型方法与Java 5.0类似，不过其不支持通配符类型。除此之外，Python、Ruby等动态语言虽然没有显式的类属设施，但却可以有效支持泛型程序设计。因为这些语言中的变量没有固定类型，所以子程序的形参也没有类型，只要该子程序中对形参使用的操作符被定义过，那么一个子程序对于任意类型的实参就能运行，这与Ada中类属设施的功能是一致的。

Ada中分别编译的思想也被广泛应用于之后出现的高级语言中，包括C#在内的.Net平台语言基本都支持分别编译的功能。在.Net平台语言中，先编译好的程序模块可以作为.jar、.class或.dll文件被其他程序调用，在编译其他模块的时候无需对所调用的模块进行再次编译，从而有效提高了软件开发效率。

Ada所提供的编用设施方便程序员向编译器发出指令，编用中的很多指令都是为提高目标代码的运行效率的编译优化而设计的。在许多高级语言中，编用通常是一行用于告诉编译器该采取什么行动的源代码。编用机制虽然不是Ada语言的原创性设计，但是为控制程序行为以及提升程序的开发和执行效率提供了灵活的手段。C++、D、PL/SQL等语言亦借鉴了编用机制。

结束语 本文从语言机制、应用及影响力等方面对Ada语言的发展进行了综合的阐述和分析。

就语言机制而言，Ada所增强的部分主要涉及如下4个方面：

1. 面向对象程序设计。Ada通过扩充新的语言设施使之具备了完整的面向对象特征，并借鉴其他语言中优秀的面向对象设计理念对语言设施逐步加以优化；

2. 并发程序设计。Ada通过扩展任务通信与同步机制、补充任务调度策略等手段增强了语言本身对并发程序设计的支持；

3. 契约式程序设计。Ada 2012通过引入契约式编程为保障程序正确性和软件可靠性提供了更有力的支撑；

4. 大型软件开发。Ada 83之后的Ada语言标准通过增加后扩库单元、扩充语言标准容器库等措施有效加强了程序的可维护性，并在一定程度上提高了程序的可扩展性。

在语言应用方面，本文从软件可再用性的角度研究了Ada语言的贡献，并讨论了Ada语言的不足、现状和展望。

就语言影响力而言，本文探讨了Ada在设计理念上对C++、Java、Modula以及动态程序设计语言发展的影响。

限于篇幅，本文只能讨论与Ada相关的主要问题，很有可能挂一漏万，乞望读者指正。

参 考 文 献

[1] The Department of Defense Common High Order Language Program. AD-A059 444. Department of Defense Requirements for High Order Computer Programming Languages "STEELMAN" [S]. 1978

[2] United States Department of Defense. Stoneman: Requirements for Ada Programming Support [S]. 1980

[3] Hoare C A R. The Emperor's Old Clothes[R]. New York, NY, USA. Communications of the ACM, 1981

[4] United States Department of Defense. ANSI/MIL-STD-1815A (ISO 8652-1987). Reference Manual for the Ada Programming Language [S]. Secaucus, NJ, New York, USA: Springer-Verlag, 1983

[5] 徐宝文. 试论高级程序设计语言的设计与评价标准[J]. 南京航空航天大学学报，1987，19(2)：114-125

[6] 徐宝文. 关于Ada语言的几点修正意见 [J]. 南京航空航天大学学报，1987，19(3)：140-143

[7] Wegner P. 麦中凡，姜静波，译. Ada的成就与不足 [J]. 计算机科学，1988(1)：56-61

[8] 徐宝文. Ada语言回顾与展望[J]. 计算机应用，1989，3：46-49

[9] 徐宝文. 军用计算机语言初论[J]. 南京航空航天大学学报，1989，21(2)：102-107

[10] 徐宝文. 论Ada对DOD国防系统软件开发标准的适应性[J]. 南京航空航天大学学报，1990，22(4)：66-71

[11] 徐宝文. ANNA语言导论 [M]. 北京：中国铁道出版社，1990：20-42

[12] 徐家福，王志坚，翟成祥. 对象式程序设计语言[M]. 南京：南京大学出版社，1992

[13] Jørgensen II J. A Comparison of the Object-Oriented Features of Ada 9X and C++ [C]// Proceedings of the 12th Ada-Europe International Conference. Ada Sans Frontières. Paris, France, June 1993

[14] Anderson C. Opening Address: Ada 9X [C]// Proceedings of the First International Eurospace-Ada-Europe Symposium on Ada in Europe. Copenhagen, Denmark, September 1994

[15] United States Department of Defense. ISO/IEC 8652:1995. Ada

Reference Manual [S]. 1995

[16] Cohen N. Ada as A Second Language (2nd Edition) [M]. New York, USA: McGraw-Hill Higher Education, 1995: 9-758

[17] Barnes J. Ada 95 Rationale: The Language-The Standard Libraries [M]. Berlin, Germany: Springer, 1995: 30-97

[18] Mangold K. Ada 95-An Approach to Overcome the Software Crisis? [C] // Ada in Europe, Second International Eurospace-Ada-Europe Symposium. Frankfurt/Main, Germany, October 1995

[19] Barbey S, Kempe M, Strohmeir A. Advanced Object-Oriented Features and Programming in Ada 95 [C] // Tutorial Proceedings on Ada's Role in Global Markets: solutions for a changing complex world. Anaheim, CA, USA, November 1995

[20] 徐宝文. 程序设计语言发展回顾与展望[N]. 计算机世界报, 1995(13)

[21] Cherry G, Crawford B. The Situation in Object-Oriented Specification and Design [C] // TRI-Ada' 96 Conference, The annual meeting of the Ada programming world. Philadelphia, Pennsylvania, USA, December 1996

[22] Cross II J, Barowski L, Hendrix T, et al. Control Structure Diagrams for Ada 95 [C] // TRI-Ada' 96 Conference, The annual meeting of the Ada programming world. Philadelphia, Pennsylvania, USA, December 1996

[23] Brosgol B. A Comparison of the Object-Oriented Features of Ada 95 and Java [C] // TRI-Ada' 97 Conference, The annual meeting of the Ada programming world. St. Louis, Missouri, USA, November 1997

[24] Oh D, Baker T. Optimization of Ada 95 Tasking Constructs [C] // TRI-Ada'97 Conference, The annual meeting of the Ada programming world. St. Louis, Missouri, USA, November 1997

[25] 徐宝文. Ada95 语言评述 [J]. 计算机研究与发展, 1997, 34(1): 53-57

[26] 徐宝文. Ada95 与面向对象的程序设计 [J]. 计算机研究与发展, 1997, 34(1): 58-65

[27] 徐宝文. Ada95 层次库结构与大型软件开发 [J]. 计算机研究与发展, 1997, 34(1): 66-71

[28] 徐宝文. Ada95 保护对象与面向数据的同步 [J]. 计算机研究与发展, 1997, 34(1): 72-77

[29] 徐宝文. 关于 Ada95 变体部分等语法的修正意见 [J]. 计算机研究与发展, 1997, 34(1): 78-80

[30] 黄曙萍, 徐宝文. Ada95———一个功能强大的军用程序设计语言 [J]. 情报指挥控制系统与仿真技术, 1997, 2: 50-52

[31] Xu B. Comments On Several Syntax Rules In Ada95 [J]. ACM SIGPLAN Notices, 1998, 33(2): 65-67

[32] Li B, Xu B, Yu H. Transforming Ada Serving Tasks into Protected Objects [C] // SIGAda'98. Proceedings of the ACM SIGAda Annual International Conference on Ada Technology. Washington, DC, USA, November 1998

[33] Burns A. The Ravenscar Profile [J]. ACM SIGAda Ada Letters, 1999, 19(4): 49-52

[34] 陈火旺, 刘春林, 谭庆平. 程序设计语言编译原理[M]. 北京: 国防工业出版社, 2000

[35] 李帮清, 徐宝文. 一种 Ada 83 服务性任务向 Ada 95 保护对象变换的方法 [J]. 软件学报, 2000, 11(6): 836-840

[36] Amey P. A language for systems not just software [C] // Proceedings ACM SIGAda Annual International Conference (SIGAda 2001). Bloomington, MN, USA, September 2001

[37] Brosgol B, Dobbing B. Real-time convergence of Ada and Java [C] // Proceedings ACM SIGAda Annual International Conference (SIGAda 2001). Bloomington, MN, USA, September 2001

[38] 王振宇, 梁先忠. Ada 软件开发技术[M]. 北京: 国防工业出版社, 2001: 21-192

[39] Amey P, Chapman R. Industrial strength exception freedom [C] // Proceedings of the 2002 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies 2002. Houston, Texas, USA, December 2002

[40] Neville M, Sibley A. Developing a generic genetic algorithm [C] // Proceedings of the 2002 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies 2002. Houston, Texas, USA, December 2002

[41] Chen Z, Xu B, Yang H. Slicing Tagged Objects in Ada [C] // Reliable Software Technologies: Ada Europe 2001, 6th Ade-Europe International Conference. Leuven, Belgium, May 2002

[42] van Lamsweerde A. Building Formal Requirements Models for Reliable Software [C] // Reliable Software Technologies: Ada Europe 2001, 6th Ade-Europe International Conference. Leuven, Belgium, May 2002

[43] Brosgol B, Dobbing B. Can Java Meet Its Real-Time Deadlines? [C] // Reliable Software Technologies: Ada Europe 2001, 6th Ade-Europe International Conference. Leuven, Belgium, May 2002

[44] Sethi R. 程序设计语言[M]. 裘宗燕, 译. 北京: 机械工业出版社, 2002

[45] Xu B, Chen Z, Zhao J. Measuring Cohesion of Packages in Ada95 [C] // Proceedings of the 2003 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies 2003. San Diego, CA, USA, December 2003

[46] Riehle R. Ada Distilled-An Introduction to Ada Programming for Experienced Computer Programmers [M]. Palo Alto, California, USA: AdaWorks Software Engineering, 2003: 33-87

[47] Weiskirchner M. Comparison of the Execution Times of Ada, C and Java[Z]. September 2003

[48] Sward R. Extracting Ada 95 Objects from Legacy Ada Programs [C] // Reliable Software Technologies-Ada-Europe 2004, 9th Ada-Europe International Conference on Reliable Software Technologies. Palma de Mallorca, Spain, June 2004

[49] Burgstaller B, Blieberger J, Scholz B. On the Tree Width of Ada Programs [C] // Reliable Software Technologies-Ada-Europe 2004, 9th Ada-Europe International Conference on Reliable Software Technologies. Palma de Mallorca, Spain, June 2004

[50] Rivas M, Miranda J, Harbour M. Integrating Application-Defined Scheduling with the New Dispatching Policies for Ada Tasks [C] // Reliable Software Technology-Ada-Europe 2005, 10th Ada-Europe International Conference on Reliable Software Technologies. York, UK, June 2005

[51] Ruiz J. Ada 2005 for deeply embedded systems [J]. Embedded Computing Design, 2006, 4(2): 41-45

(下转第 38 页)

Conference on Supercomputing, ICS'08. Island of Kos, Greece: Association for Computing Machinery, 2008

[10] Collange S, et al. Barra: A Parallel Functional Simulator for GPGPU. in Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)[C]// 2010 IEEE International Symposium on. 2010

[11] Volkov V, Demmel J W. Benchmarking GPUs to tune dense linear algebra[C] // 2008 SC-International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008. Austin, TX, United states: IEEE Computer Society, 2008

[12] Zhang Y, Cohen J, Owens J D. Fast tridiagonal solvers on the GPU[C]//2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10. Bangalore, India: Association for Computing Machinery, 2010

[13] Goddeke D, Strzodka R. Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid [J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 23(1): 22-32

[14] Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors[C] // SC'09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing. Nov. 2009, 18: 1-11

[15] Choi J W, Singh A, Vuduc R W. Model driven autotuning of sparse matrix-vector multiply on GPUs[C]//Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010). ACM, Jan. 2010: 115-126

(上接第 15 页)

[52] Barnes J. High Integrity Software: the SPARK approach to Safety and Security [M]. London, UK: Addison-Wesley, 2006: 3-53

[53] Kaiser C, Pradat-Peyre J, Évangelista S, et al. Comparing Java, C# and Ada Monitors queuing policies: a case study and its Ada refinement [J]. ACM SIGAda Ada Letters, 2006, 26(2): 23-37

[54] Klein J. Use of Ada in Lockheed Martin for air traffic management and beyond [C] // Proceedings of the 2006 Annual ACM SIGAda International Conference on Ada. Albuquerque, New Mexico, USA, November 2006

[55] 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

[56] Dewar R. Ada 2005 & high integrity systems [C]//Proceedings of the 2006 Annual ACM SIGAda International Conference on Ada. Albuquerque, New Mexico, USA, November 2006

[57] WG9. ISO/IEC 8652: Ada Reference Manual(Ed. 3)[S]. 2007

[58] Burns A, Wellings A. Concurrent and Real-Time Programming in Ada 2005 [M]. Cambridge, London, UK: Cambridge University Press, 2007: 451-453

[59] Sward R. Using Ada in a Service-Oriented Architecture [C]// Proceedings of the 2007 Annual ACM SIGAda International Conference on Ada. Fairfax, Virginia, USA, November 2007

[60] Barnes J. Ada 2005 Rationale: The Language-The Standard Libraries [M]. Berlin, Germany: Springer, 2008: 31-237

[61] Tokar J. 30 years after steelman, does DoD still have a software crisis? [C]//Proceedings of the 2008 Annual ACM SIGAda International Conference on Ada. Portland, OR, USA, October 2008

[62] Brosgol B. From strawman to Ada 2005: a socio-technical retrospective [C] // Proceedings of the 2008 Annual ACM SIGAda International Conference on Ada. Portland, OR, USA, October 2008

[63] Martínez P, Drake J, Pacheco P, et al. An Ada 2005 Technology for Distributed and Real-Time Component-Based Applications [C] // Reliable Software Technologies-Ada-Europe 2008, 13th Ada-Europe International Conference on Reliable Software Technologies. Venice, Italy, June 2008

[64] Sebesta R. Concepts of Programming Languages[M]. 9th International Edition. Hong Kong: Pearson Education, 2009

[65] Schonberg S. Ada 2012 Intrim Report [C]//Proceedings of the 2010 Annual ACM SIGAda International Conference on Ada. Fairfax, Virginia, USA, October 2010

[66] Sward R. The Rise, Fall and Persistence of Ada [C]//SIGAda'10 Proceeding of the ACM SIGAda Annual International Conference on SIGAda. USA, October 2010: 71-74

[67] Rosen J. Developing a Profile for Using Object-Oriented Ada in High-Integrity Systems [J]. ACM SIGAda Letters, Fairfax, Virginia, USA, 2010, 31(1): 9-10

[68] Moore B. Parallelism Generics for Ada 2005 and Beyond [C]// SIGAda'10 Proceeding of the ACM SIGAda Annual International Conference on SIGAda. USA, October 2010: 41-52

[69] Barnes J. A Brief Introduction to Ada 2012 [R]. Edinburgh, UK: The GNAT Pro Company, 2011

[70] Saez S, Terrasa S, Crespo A. A Real-Time Framework for Multiprocessor Platforms Using Ada 2012 [C] // Reliable Software Technologies-Ada-Europe 2011-16th Ada-Europe International Conference on Reliable Software Technologies. Edinburgh, UK, June 2011

[71] Chapman R, Jennings T. OOT, DO-178C and SPARK [C]//Reliable Software Technologies-Ada-Europe 2011-16th Ada-Europe International Conference on Reliable Software Technologies. Edinburgh, UK, June 2011

[72] Rosen J-P. Object Orientation in Critical Systems: Yes, in Moderation-Position Paper for the DO178C and Object-Orientation for Critical Systems Panel [C] // Reliable Software Technologies-Ada-Europe 2011-16th Ada-Europe International Conference on Reliable Software Technologies. Edinburgh, UK, June 2011

[73] Tokar J, Jones F, Black P, et al. Software vulnerabilities precluded by spark [C] // Proceedings of the 2011 Annual ACM SIGAda International Conference on Ada. Denver, Colorado, USA, November 2011

[74] WG9. ISO/IEC 8652: 2012 (E). Ada Reference Manual[M]. December 2012

[75] Schonberg E, Pucci V. Implementation of a simple dimensionality checking system in Ada 2012[C] // Proceedings of the 2012 ACM conference on High integrity language technology. New York, NY, USA, December 2012

[76] Ruiz J, Comar C, Moy Y. Source Code as the Key Artifact in Requirement-Based Development: The Case of Ada 2012[C]//17th Ada-Europe International Conference on Reliable Software Technologies. Stockholm, Sweden, June 2012

[77] Tempelmeier T. Teaching Concepts of Programming Languages' with Ada[C] // 17th Ada-Europe International Conference on Reliable Software Technologies. Stockholm, Sweden, June 2012