编程方法论的思考

芭芭拉·利斯科夫（Barbara Liskov）
麻省理工学院

关键词：编程方法论　模块化设计　面向对象编程

译者按：2008年，芭芭拉·利斯科夫（Barbara Liskov）因其对编程语言和系统设计作出的贡献而获得ACM图灵奖，有人评论道："她凭什么获奖？每个人都知道这个。"这条评论恰恰展示了芭芭拉的工作所产生的深远影响。20世纪70年代困扰无数工程师的问题在30年后成为了人们的常识，其应用程度之广泛甚至让人很难去想象，在其没有出现之前构建一个有效的程序是多么困难。本文基于芭芭拉在CNCC2021上的特邀报告整理而成，力图回溯编程方法论的演进过程。

软件危机

1970年左右，我在MITRE公司完成Venus操作系统后，一项新的任务交给了我：调查软件危机。现在大家可能对这个词很陌生，但在当时，人们并不懂得如何构建有效的程序。从20世纪60年代到80年代，人们都可能读到过类似的报道——一个公司为开发某个系统投入了数百万美元和数百人年，但到了最后这个系统还是无法运行。这是当时一个亟待解决的重要问题。

面对这样一个陌生的领域，我开始深入研究文献，并发现大量论文聚焦在一个被称为"编程方法论（programming methodology）"的领域。这些研究关注的是两个既彼此独立又相互关联的问题：如何设计程序？如何构建程序？这两个问题都在谈论模块化（modularity），认为模块化的目的是把程序分成几个相对独立的部分，以便分而治之。但模块（module）究竟是什么呢？这一点在当时并没有很清晰的定义。

在这里，我要谈谈大卫·帕纳斯（David Parnas）的一篇关于规范（specification）的论文[1]。该论文指出无论模块的接口是什么，它必须体现三个点：（1）该模块向系统其余部分展示的所有行为；（2）该模块与其他模块之间的所有连接；（3）其他模块对该模块所做的所有假设。这篇论文打破了人们当时认为模块可能是程序的想法。因为模块之间还有很多其他的联系，人们只需要关心模块的参数和结果。与此同时，帕纳斯还发表了另一篇论文，文中写道："我们知道我们想要模块化，但我们不知道模块是什么。"这句话非常直观地体现了当时人们对"模块"这个概念的普遍认知。

由于计算机的快速发展，到20世纪60年代时，机器的计算能力相比之前提高了几个数量级，软件系统的规模越来越大，系统复杂度也随之提高。软件项目的开发周期、代码质量要求、成本都发生了质的变化。原先开发小型软件的方式（工作室方式）不再适用于大型系统的开发，此时迫切需要改进软件开发方式，提高软件开发效率，降低软件维护成本。在这样的背景下，北大西洋公约组织于1968年提出了软件危机（software crisis）。IBM System/360系统是当时的一个典型案例，它是第一个超大型的软件项目，使用了约

图 1　芭芭拉在 CNCC2021 上作特邀报告

1000 名程序员，其项目经理弗雷德里克·布鲁克斯（Frederick P. Brooks, Jr.）事后说到，他在管理这个项目的时候，犯了一个价值数百万美元的错误。弗雷德里克根据该系统的开发经验，编写了被誉为软件领域圣经的《人月神话：软件项目管理之道》。

为了应对软件危机，人们开始谈论使用模块来设计和构建程序，但是却不知道如何清晰地定义模块，例如有论文写着“一个模块不能超过一千行代码”。大家只明白他们需要一种控制程序开发过程的方法，以便分工合作。—译者注

编程方法的演进之路

当下，我们认为程序是模块的集合，每个模块都有一系列用规范描述的接口。而规范是对模块预期行为的描述，它可以是用正式的数学语言编写的，也可以是非正式的。但不管这个规范是怎么编写出来的，它对模块行为的描述及其具体实现是相互独立的。例如对于一个排序程序来说，输入的数组和返回的结果应包含相同的元素，只是按照顺序重新排列了一遍。

对于模块，正确的实现是指提供了规范中指定的行为。所以当你审视一个模块的代码时，可以使用规范对代码进行推理。代码可能像排序程序一样很简单，也可能更复杂。一个模块可能会调用其他模块，即便如此，仍然可以通过本地推理（local reasoning）的方法来依次理解，因为你只需要关注某一模块。通过这种方式，我们实际上降低了开发一个复杂软件的难度。

回到 1970 年，我们了解了模块化的优点，了解了局部推理的重要性，但并不知道它具体是什么。通过模块化，一个团队中的成员就可以同时独立开发不同的模块。这样是便于修改的，一旦程序中的某些模块运行异常，可以用另外一个实现来替代它，整个系统的运行不会受到影响。

全局变量搞定一切——早期的结构化编程

在阅读了帕纳斯的论文[1]和其他文献后，我意识到自己在开发 Venus 操作系统时就发明了一种模块化机制，虽然当时并没有专门研究设计方法或编程方法，只是聚焦于能否构建出有效的软件。那时，我试图规避使用全局变量来组织代码的问题。

1958 年诞生的 ALGOL 语言引入了块的概念，被认为是最早的结构化编程语言。ALGOL 使用 begin 和 end 关键字表示代码的开始和结束，两个关键字中间的语句称为代码块，一个代码块可以嵌套在其他代码块中。在这种形式下，尽管内部块可以访问外部块的数据，但外部块却无法访问内部块的私有数据。所以为了实现块与块之间的通信，常见的方法是大量使用全局变量。——译者注

分区——模块化机制的雏形

我在设计 Venus 操作系统时使用了一种被称为

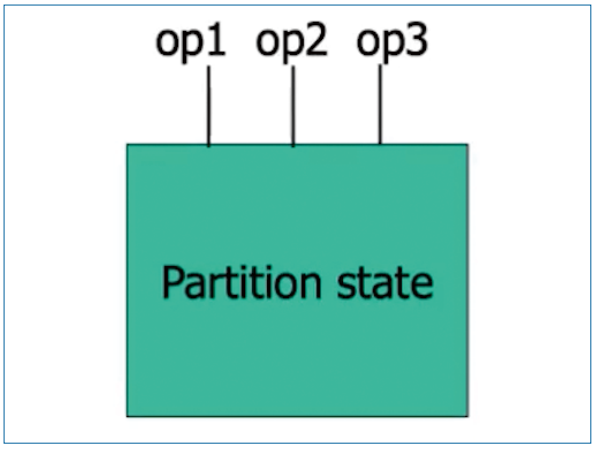

图 2　分区

“分区（partitions）”的概念。在 Venus 系统中，我将系统中已经存在的所有全局变量划分到不同模块中，每个模块都拥有一部分全局状态，并且禁止其他模块访问这些状态。但是同时，模块之间可以通过一系列方法与这些状态进行交互。这个想法的有趣之处在于它比其他人提出的模块化机制更有条理，因为这不仅仅是黑盒，它还有一个实际的接口结构（见图 2）。所以我将其写了下来，并发表在 ACM 操作系统原理大会（SOSP）上 [2]。

我发表的关于 Venus 操作系统的论文获了奖。在 SOSP 的演讲后，我被邀请到麻省理工学院（MIT）工作。这对我来说是一个很好的机会，因为编程方法是我热衷的研究方向。

Venus 操作系统是芭芭拉博士毕业后在 MITRE 的第一个工作，她用微编程（microprogramming）设计并实现了一个机器架构 Venus，并且在这个机器架构上构建了一个小型分时操作系统 Venus。——译者注

从分区到抽象数据类型

1972 年秋天，我在思考如何让程序员更容易理解论文中提到的想法。当程序员阅读一篇论文时，可能会觉得“哇，是的，就应该这么做，这个方法看起来很完美”。但是当程序员尝试将论文中的想法应用到自己的程序中时，一切都崩溃了，总觉得缺少了一些东西。我试图弄清楚缺少的究竟是什么。

几个月后我有了一个绝妙的解决办法—可以将分区联系到抽象数据类型（Abstract Data Type，ADT）。正如我们今天所知，抽象数据类型是指通过一系列方法为用户提供接口的对象，我们可以通过构造函数来创建对象，并通过一些方法与对象进行交互。所以当你观察一个分区时，就可以知道发生了什么。

我们通过创造抽象概念来设计模块，从而将模块化与设计联系起来。当我们考虑文件系统、设备驱动程序和 I/O 模块时，它们不仅仅是程序，还是一堆小模块的集合，数据抽象是一个比程序更大的概念。程序员理解数据类型和编程语言，他们明白这些实际上是抽象的，无法全部通过计算机的硬件功能来实现，只能通过软件的功能来达成。而且他们也能够通过编程来创造新的数据类型，所以这似乎是一个非常好的想法，但是要让它落地则需要编程语言的支持。正如我刚才所说，当时没有提供抽象数据类型的编程语言，所以我开始研究这个点。

我和史蒂夫·齐勒（Steve Zilles）一起工作，他是 MIT 的研究生也是 IBM 的员工。我们紧张地工作了几个月，其中一项工作就是再次阅读我们可以拿到的所有论文。在编程语言方面，我们已了解当时存在的大多数语言，诸如 Fortran、COBOL、PL/I，同时更加清楚 ALGOL 在学术上的重要意义，我们进行了充足的技术储备。

詹姆斯·莫里斯 (James H. Morris) 写的论文“Protection in Programming Languages”[3] 对我的工作影响很大。他在论文里提到了一个问题：“为了使局部推理起作用，我们必须遵循哪些规则？”正如我之前所说，局部推理对模块化非常重要，但我们并没有真正理解规则是什么。他在这篇论文中提到“模块外的代码不得修改由模块管理的数据”，这是很重要的一点，因为一旦外部代码可以修改模块内部的数据，你显然不可能通过本地推理验证模块代码是否正确。他还指出“基于模块的可修改性，外部的代码甚至不应该观察到内部的数据”。如果人们试图重新实现该模块，不需要对模块外的代码进行任何变动，将旧模块替换为新模块即可。通过每

天的阅读和讨论，1973 年夏末，我们已经对这个编程语言有了一个很好的构想，所以撰写了论文“Programming with Abstract Data Types”[4]。

这篇论文提出了数据抽象，以及对一种可以提供抽象数据类型概念的编程语言的需求。论文中提出将数据类型视为一组对象，它还有一系列操作，分别用于创建对象和使用对象，这些操作应是人们与模块交互的唯一方式。我们在论文中阐明：我们希望通过严格的检查来强制执行此规则。这篇论文包含了一种编程语言的构想。接下来，我决定设计并实现这个语言。这项工作主要与当时加入我小组的三位研究生（Russ Atkinson、Craig Schaffert 和 Alan Snyder）一起完成[5]。

这样做的意义是什么呢？首先，编程语言是一个数学对象，这意味着人们要针对正在发生的事情制定非常精确的规则。很多时候人们遇到麻烦的一个原因就是没有精确的规则。此外，编程语言是一种工具，只有人们实际使用它的时候，才能知道这个工具是否能解决问题。现在来看这似乎是顺理成章的，但在当时，能否拥有一种支持抽象数据类型、易使用且表现力强的编程语言是被怀疑的。当时的担忧之一是，随着编程语言的层次越来越高，程序员可能会无法实现他们想要的某些功能。当然，这个编程语言的性能必须够用。所以我们需要考虑这几个方面，以便拥有一个实用的工具。

文献 [4] 提出，对编程语言或系统来说，有效的保护机制至关重要，论文还从对象、局部变量、内存、访问控制等各层面来探讨保护原则。前文提到的抽象数据类型是指一个数学模型以及定义在该模型上的操作。它能够使人们独立于程序来理解数据结构的特性和作用。——译者注

从抽象数据类型到 CLU

我们花了很长时间设计 CLU，尝试了很多不同的方式。首先，我们必须想出一种实现抽象数据类型的技术，这就是我们所说的簇（cluster）。实际上，CLU 的名字来源于 cluster 的前三个字母。簇与我们今天使用的类（class）很相似，但还包含其他方面的内容。数据抽象通常是集合（set）、列表（list）或树（tree）这样的集合，人们经常需要遍历一个集合并操纵里面的元素，比如打印一个集合中的所有元素。因此，我们必须找到一种简单的遍历机制，在不破坏集合中元素的同时对它们进行遍历。当时，编程语言中几乎没有异常处理机制，因为我们聚焦于正确性和规范，所以我们必须弄清楚异常处理机制是什么。通常情况下，程序无法对其合法参数的所有元素执行完全相同的处理，假设要从集合中选择一个元素，如果集合为空，人们会怎么做？众所周知，错误的来源通常远离产生错误的地方，所以当遇到这样的异常情况时，我们需要一个通用的异常处理机制。最重要的是，当时的编程语言不支持我们需要的多态或泛型，所有的语言都只提供几种内置类型。因此，如果要编写一个对整数数组进行排序的程序，则必须编写另一个程序对字符串数组进行排序。人们每发明一种数据类型，都必须重新实现所有代码，所以我们需要一个通用的机制来规避这个问题。此外，数据类型本身必须是通用的。所以我们发明了 CLU 这个编程语言，并在 1977 或 1978 年左右将其推广。后来我们有了一个可以运行的 CLU 编译器。很快这个编程语言就有了一批用户，我们证明了人们可以拥有一种具有数据抽象的语言，并取得了良好的反响。

CLU 远远领先于它所处的时代。这不仅仅因为它有数据抽象而其他编程语言没有，而且 CLU 中的模板、异常处理机制、迭代器和多赋值等在今天看来很普遍的概念和功能，在当时的编程语言中并不存在。因此，CLU 也启发着之后的编程语言的设计，如 C++、Java 和 Python 等。——译者注

CLU 之后

发明了 CLU 之后，我试图弄清楚下一步要怎么做。我读了互联网的发明者之一鲍勃·卡恩（Bob Kahn）的一篇论文，在这篇论文中，卡恩谈到了他对分布式计算的构想。在分布式计算中，人们可以调用同一个网络中的所有程序，不管具体是在哪一台机器上运行，但当时没有人知道这一点要如何实

现，而这正是我所要解决的问题。在 CLU 的设计中我没有考虑并发的问题，这是为了保证设计处于可控状态。但在分布式领域中，我可以重新考虑并发机制。所以在我接下来的职业生涯中主要从事分布式计算的工作。

但我并没有失去对编程方法论的兴趣。实际上我在 MIT 设计一门课程时，在模块化程序设计、正确性推理等方面也做了很多工作。这其中的大部分是与约翰·古塔格（John Guttag）共同完成的，我们发表了几篇关于这项工作的论文，并编写了一本教科书 *Abstraction and Specification in Program Development* [6]，使用 CLU 作为编程语言。后来我们写了一本使用 Java 作为编程语言的书。

芭芭拉在开发 CLU 语言时并没有考虑并发性，因为当时 CLU 的功能已经足够多了，并且考虑并发会分散注意力。后来，当她开始从事分布式计算工作时，就需要再次考虑并发这个问题。在 70 年代后期，虽然芭芭拉开始研究分布式计算，但她一直在思考“如何推理（reason about）抽象数据类型的正确性”“如何为抽象数据类型编写规范”等问题。——译者注

从 CLU 到面向对象编程

我今天还要谈论我在编程方法论方面所做的一些其他工作——类型层次结构（type hierarchy）。1987 年，我被邀请在面向对象编程语言会议（OOPSLA）上作一个主题演讲，我可以借机去了解 SmallTalk 和其他衍生语言有什么新的进展，因为我要考虑面向对象编程，它混合了抽象数据类型和继承等思想。我阅读了关于 SmallTalk 和一大批提供了继承机制的文献。继承是作为一种技术实现开始的。当人们已经有一个承担某项功能的类，但还想要一个子类来承担相似但不相同的功能时，可以通过借用超类来实现这个功能。虽然许多论文谈论的具体内容略有不同，但它们都将其归纳为类型层次结构。我们的问题是，当编写需要超类对象的代码时，它可能会在参数中接收子类的对象。那些文献的作者试图弄清楚这意味着什么，因此，他们将子类的对象视为超类对象的子类型。

我发现那些论文的作者对于类型层次结构不是很理解。我记得有一篇论文说“堆栈和队列都是彼此的子类型”，这显然不对。设想如果你编写一些需要堆栈的代码，但最后收到的却是一个队列，即你期望的是后进先出（Last In First Out，LIFO）的行为，但你得到的却是先进先出（First In First Out，FIFO）的行为，你的代码则不太可能正常工作。

在 1987 年 7 月的演讲中，我对行为子类型（behavioral subtyping）做了一个简单的定义。如果使用超类方法，子类型对象的行为应该与超类型对象的行为一样。在这种约束方式下，子类型必须与超类型相同。1988 年，*ACM SIGPLAN Notices* 邀请我将其写下来。后来，我与周以真（Jeannette Wing）共同对工作进行了梳理，并对正在发生的事情提出了正式定义。

这后来被称为 Liskov 替换原则（Liskov Substitution Principle）。这是一个非常重要的概念，从“behavioral subtyping”的语义上来看，虽然里面有“行为”这个词，但我真正谈论的是规范。如果人们考虑模块化，就可以根据这些规范来证明一个模块支持着另外一个模块，最后，人们就可以证明这个实现运行正常。

由于芭芭拉发现很多人并未真正的理解超类和子类的意义，所以在这次演讲中她给出了一个非正式的定义：子类应该在使用超类方法的时候表现得如同超类一样 [7, 8]。——译者注

总结

我今天一直在谈论模块化以及该概念的发展历程。当我获得 2008 年 ACM 图灵奖时，我发现我的学生并不了解在有数据抽象这个概念之前编程是如何进行的。我丈夫每天都在网上浏览关于我获得图灵奖的评论。他看到了一条看上去不那么友好的评论：“她凭什么获奖？每个人都知道这个。”说此话的人本意并不是批评我，实际上这是一种巨大的赞美，因为基于我和我团队的工作，基于在编程语言

和编程方法领域的共同努力，我们探索出了今天构建软件的方式。这就是为什么当下编程的方式是基于抽象的模块化。

芭芭拉在 CNCC2021 上的特邀报告，带领我们回顾了编程方法的演进过程。从最早的不理解什么是“模块”，演进到模块化的面向过程编程，再到提出抽象数据类型和 CLU 编程语言（这一阶段更像是从面向过程编程到面向对象编程的过渡），再到最后的面向对象编程，每一阶段所出现的新编程思想和方法，都让编程这件事情变得更高效、更容易管理。——译者注 ■

作　者：

芭芭拉·利斯科夫（Barbara Liskov）

2008 年 ACM 图灵奖获得者。美国国家工程院院士、美国艺术与科学院院士。麻省理工学院教授，主管研究工作，计算机科学与人工智能实验室编程方法论小组负责人。

译　者：

郭　静

CCF 学生会员。中国科学院计算技术研究所博士研究生。主要研究方向为云计算系统资源管理。
guojing@ict.ac.cn

胡存琛

CCF 学生会员。中国科学院计算技术研究所博士研究生。主要研究方向为云计算资源调度。
duihuhu@gmail.com

包云岗

CCF 高级会员，*CCCF* 前编委。中国科学院计算技术研究所研究员。主要研究方向为计算机系统结构，包括数据中心体系结构、处理器芯片敏捷设计、开源芯片生态等。baoyg@ict.ac.cn

参考文献

[1] Parnas D L. Information distribution aspects of design methodology[C]// *Proceedings of IFIP Congress 71. North-Holland*, 1971: 339-344.

[2] Barbara H. Liskov. The design of the Venus operating system[J]. *Communications of the ACM*, 1972, 15(3): 144-149.

[3] Morris J H. Protection in programming languages[J]. *Communications of the ACM*, 1973, 16(1): 15-21.

[4] Liskov B, Zilles S. Programming with abstract data types[J]. *ACM SIGPLAN Notices*, 1974, 9(4): 50-59.

[5] Liskov B, Snyder A, Atkinson R, et al. Abstraction mechanisms in CLU[J]. *Communications of the ACM*, 1977, 20(8): 564-576.

[6] Liskov B, Guttag J V. *Abstraction and Specification in Program Development*[M]. MIT Press and McGraw Hill, 1986.

[7] Liskov B. Data abstraction and hierarchy[J]. *Addendum to the Proceedings of OOPSLA '87, SIGPLAN Notices*, 1988, 23, (5): 17-34.

[8] Liskov B, Wind J. A behavioral notion of subtyping[J]. *ACM Transactions on Programming Languages and Systems*, 1994, 16(6):1811-1841.

（本文根据 CNCC2021 特邀报告整理而成）