编程系统的技术维度

Joel Jakubovic[a]、Jonathan Edwards[b] 和 Tomas Petricek[a, c]
a 英国坎特伯雷肯特大学 b 独立研究
者 c 捷克布拉格查理大学

摘要 编程远不止是用编程语言编写代码。编程通常是在有状态环境中，通过图形用户界面与系统交互来完成的。然而，这个广阔的可能性空间缺乏用于导航的共同结构。有关编程系统的研究尚未形成连贯的研究体系，因此难以在既有工作基础上改进并推动该领域的发展。

在计算机科学领域，人们为比较编程语言做了大量工作，也提出了许多相关论述，但编程系统却没有类似的理论；我们认为，编程系统同样值得拥有一套理论。
我们提出一个技术维度框架，用以捕捉编程系统的内在特征，并为对其进行概念化和比较提供方法。

我们通过考察过去有影响力的编程系统，并审视其设计原则、技术能力和用户交互方式，来识别技术维度。技术维度体现了可独立研究、比较和发展的特征。这使我们能够以一种可交流、可进行建设性讨论的方式谈论编程系统，而不必完全依赖个人印象。

我们的框架源自对过去编程系统的定性分析。我们概述了两种具体的框架使用方式。首先，我们展示如何用它分析一个近期开发的新型编程系统。接着，我们用它来识别编程系统设计空间中一个尚未探索、颇有意义的点。

许多研究工作致力于构建更易用、非专业人士也能使用、可塑性更强和／或功能更强大的编程系统，但这类工作彼此割裂。它们缺乏形式化，受作者的个人愿景指引，因此只能依据个人使用体验来评估和比较。通过为更系统的研究奠定基础，我们可以帮助编程系统研究者终于站在巨人的肩膀上。

ACM CCS 2012

- **软件及其工程** → **集成与可视化开发环境**；**以人为中心的计算** → 交互式系统与工具； 人机交互设计
- 与评估方法；

关键词 编程系统、维度、设计、框架、分析

《编程的艺术、科学与工程》

提交于 2022年9月30日**发布于**
2023年2月15日

DOI 10.22152/programming-journal.org/2023/7/13

© Joel Jakubovic、Jonathan Edwards 和 Tomas Petricek 本作品采用“CC BY 4.0”许可证授权。载于《编程的艺术、科学与工程》，第7卷，第3期，2023年，第13篇文章；59页。

> 系统化的呈现会将思想从孕育它们的土壤中抽离出来，并以一种人为的模式将其排列。
>
> 《科学的暴政》保罗·费耶阿本德

> 据说，反讽能让艺术家在沉浸于一个自己持批判态度的社会文化语境中时，仍继续进行创作。
>
> 反讽：或后现代主义建筑的自我批判性晦涩埃马纽埃尔·珀蒂

1 引言

人们开发了多种软件来支持编程。经典形式由编程语言、用于输入源代码的文本编辑器，以及将其转换为可执行程序的编译器组成。这种形式的各种实例在语言的语法和语义，以及编译器或运行时环境中的实现技术方面有所不同。自图形用户界面（GUI）问世以来，编程语言可以嵌入图形环境中，而图形环境日益决定程序员使用该语言的方式——例如，直接支持调试或重构。除此之外，GUI 的兴起还促成了多样化的可视化编程形式，包括可视化语言和基于 GUI 的终端用户编程工具。

本文主张将关注点从编程语言 转向更一般的“支持编程的软件”这一概念——换言之，即编程系统。

定义（编程系统）。 编程系统是一个集成且完整的工具集合，足以用于创建、修改和执行程序。它包括用于组织程序和数据的表示法、用于运行和调试程序的功能，以及用于执行所有这些任务的接口。此外，还可能包含测试、分析、打包或版本控制功能。表示法包括编程语言，接口包括文本编辑器，但并不限于此。

这一概念涵盖经典编程语言及其编辑器、调试器、编译器和其他工具。不过，它的范围也有意设得足够宽泛，既包括 Smalltalk 这类基于图像的编程环境、UNIX 这类操作系统，也包括 HyperCard 这类超媒体创作系统，以及我们将提到的其他各种实例。

1.1 问题：系统缺乏系统化框架

无论是在编程研究社区还是工业界，人们对更广泛的*编程系统*形式都越来越感兴趣。研究人员正在探讨*编程体验*和*实时编程*等主题；这些主题要求我们不仅考虑*语言*，还要考虑特定系统的其他方面。与此同时，商业公司正在构建新的编程环境，例如 Replit [78]，以及 Dark [56] 和 Glide [30]这类低代码工具。然而，这些主题仍处于主流编程研究的边缘。尽管*编程语言*已是一个成熟的概念，人们可以用共同的词汇对其进行分析和比较，但对于范围更广的*编程系统*，尚不存在类似的基础。

编程领域的学术研究也受制于缺乏共同词汇这一问题。我们或许能够对编程语言进行全面评估，但一旦将交互或图形纳入其中，除了主观的“酷炫程度”之外，要进行评估就困难重重。[1] 此外，在设计新系统时，灵感往往来自少数几个独立的思想来源，例如 Smalltalk 这类有影响力的早期系统、电子表格这类可编程终端用户应用，或 Bret Victor [90] 那样具有启发性的演示。

这些涌现的想法难以彼此关联，因而未能形成一个扎实的研究成果体系。用于研究编程系统的方法缺乏编程语言研究方法那样严谨的结构。这些方法往往依赖单例示例，它们虽然展示了作者的想法，却不足以用于将新想法与他人的工作进行比较。因此，人们难以在既有成果的基础上继续推进，也就难以提升技术水平。

研究*编程系统*不仅仅是选取一种编程语言，再考察围绕它构建的工具。这意味着转向一种*范式转变*后的视角，而这种视角至少在一定程度上与语言的视角*不可通约*。研究编程语言时，一切重要的东西都在程序代码中；研究编程系统时，一切重要的东西都在程序员与系统之间的*交互*中。正如 Gabriel [28], 所记录的那样，从*语言*的视角看*系统*，会让人无法思考那些源自与系统交互、却未体现在语言中的概念。因此，我们必须谨慎行事。正如我们将看到的，当我们把 Lisp 称为一种编程系统时，所指的与一种括号繁多的编程语言截然不同！

[1] Olsen [68] 对用户界面系统领域中的同一难题进行了分析。有趣的未来工作可以是对编程系统相关出版物进行详细分析，以深入理解这一问题。一个显著特点是，出版物往往介绍新系统（或其部分）。LIVE 2020 和 2021 年研讨会的论文中，分别有 5/6 和 6/7 篇属于这种情况 [39, 37]。相比之下，编程*语言*领域的出版物通常探讨许多语言共同关注的具体问题。

1.2 贡献

我们提出一种通用语言，作为推动编程系统研究迈向更进步方向的第一步。我们的技术维度 集合旨在沿着若干具体的“坐标轴”，拆解对系统的整体性看法。这些维度界定了一系列可能的设计选择，并以设计空间中的两个极端点作为特征。它们并非定量指标，但可以通过将系统定位在共同的坐标轴上进行比较。我们无意将极端点视为“好”或“坏”的设计；我们认为，任何一种定位都是设计权衡的结果。在这一框架发展初期，我们鼓励先就系统的描述达成共识，再处理任何规范性判断。

这些维度构成了编程系统设计空间的一幅地图（图1）。过去和现在的系统将作为地标；地标足够多时，我们或许就能揭示尚未探索或被忽视的可能性。迄今为止，这一领域还未能建立良性的反馈循环；从业者很难将自己的工作置于其他人的工作背景中，以便后续工作能够在此基础上改进。我们的目标是为编程系统研究奠定基础，从而推动这类发展。

本文旨在作为关于技术维度框架当前状态的参考资料，供人使用，而非通读。我们详细介绍各个维度，但建议读者初次阅读时略读细节。之后，我们建议读者重新审视那些似乎与自己熟悉的具体系统相关的维度。本文的主要贡献如下：

1. 在第3节中，我们阐述编程系统的定义，并回顾过去具有里程碑意义的编程系统。这些系统将在本文中作为示例，也有助于界定我们对编程系统的理解。2. 我们详细介绍技术维度，并将相关维度归为若干集群：交互、符号表示、概念结构、可定制性、复杂性、错误和可采纳性。对于每个维度，我们都给出示例，以展示其坐标轴上取值的范围。我们希望这份目录能作为按需查阅的参考资料，而不是从头到尾通读的内容，因此建议初次阅读时浏览目录。3. 在第5节中，我们概述使用技术维度框架的两种方式。在第5.1节中，我们用它评估一个近期颇有意思的编程系统；在第5.2节中，我们用它找出设计空间中一个尚未探索的位置，并构想一种潜在的新型编程系统。

2 相关工作

虽然我们确实提出了一些新想法，但我们的贡献之一是将各种已有的概念整合到一个统一的框架下。这项工作分布在

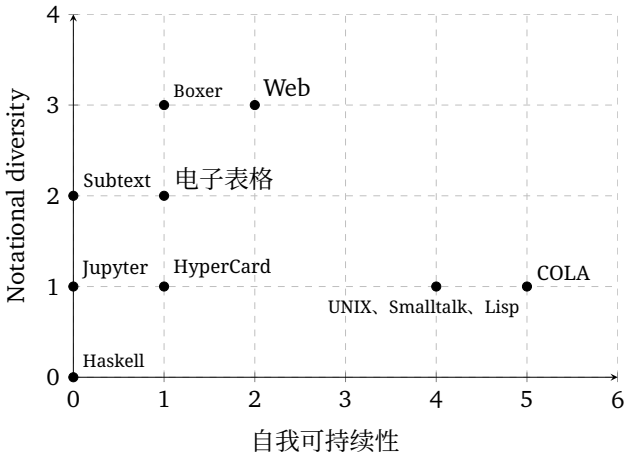


图1 可能系统空间的一个二维切片，将在第5.2节中进一步考察。每个坐标轴上的数值评分均通过附录A所述的方法系统地生成。尽管这些结果看起来合理，但并非定论，因为未来工作中还可以大幅改进这一方法（见第A.3节）。

不同领域，但每一部分都与编程系统相关，或聚焦于编程系统可能具有的某项特征。

从语言到系统。 我们的方法介于狭义的编程语言研究与广义的、将编程视为社会政治和文化议题的研究之间。我们的编程系统概念在范围上属于技术层面，不过我们承认，技术层面往往具有重要的社会影响，“可采纳性”维度（第4.7节）便是一个例子。这与 Tchernavskij [88] 或软件研究 [25]中更侧重社会政治的研究形成对比。我们的观点与 Kell 对 UNIX、Smalltalk 及操作系统整体的概念化 [45], 有所重叠，并且我们确保 UNIX 在我们的框架中占有一席之地。

较狭义的*编程语言* 与较广义的*编程系统* 之间的区别更为微妙。Richard Gabriel 指出，在计算机科学领域，研究重点曾发生过一次从“系统”转向“语言”的无形范式转变 [28], ，这一观察也为我们在此所做的区分提供了依据。这一变化的一个结果是，*语言* 通常会在脱离任何特定实现的情况下得到形式化规范，而*系统* 则难以进行形式化规范，往往*由* 某种实现来定义。我们认为，编程语言实现只是可能的系统空间中的一个小小区域，至少就交互和符号表示而言是如此。因此，我们所说的是语言的*交互式编程系统* 特性，例如文本编辑和命令行工作流。

编程系统研究。 近期，随着新型编程工具的出现，人们重新燃起了对编程系统的兴趣：

- Jupyter [51] 等计算笔记本将代码片段与文本和视觉输出结合起来，助力数据分析。它们由有状态的“内核”驱动，并以交互方式使用。“低代码”终端用户编程系统主要通过图形用户界面支持应用程序开发。Coda [12], 就是一个例子，它结合了表、公式和脚本，让非技术人员也能构建“文档即应用程序”。Dark [56], 等领域专用编程系统则宣称能为云 API 服务提供“整体式”的编程体验，其中包括一种语言、一个直接操纵编辑器，以及近乎即时的构建与部署。即使是使用传统工具进行通用编程，Replit [78] 等系统也展现了将所需的各种语言、工具和用户界面整合为无缝体验的优势；用户可通过浏览器使用这一体验，无需进行任何设置。

遵循编程系统视角的研究可以在多个研究场合中找到，包括 UIST[2] 和 VL/HCC[3] 等人机交互会议。然而，这些场合中的研究往往更强调用户体验，而非技术描述。编程系统常在 LIVE 和 PX 等研讨会中展示[4] 。然而，这些场合中的研究往往局限于作者的个人视角，也难以与其他系统进行比较，存在前文所述的问题。

本文中随处可见系统的具体示例。近期启发我们提出部分维度的系统包括: Subtext [17],, 它将代码及其实时执行整合在同一个可编辑的表示中; Sketch-n-sketch [38], , 它能够通过直接操作输出合成代码; Hazel [69], , 一种实时函数式编程环境，借助带类型空洞支持执行不完整或类型错误的程序; 以及 Webstrates [50], , 它通过扩展网页实现状态的实时共享。

已知特征。已有多个项目致力于识别编程系统的特征。有些项目围绕单一特征展开，例如活跃度级别 [87], , 或多元性与可交流性 [46]。另一些则提出了一整套特征。《编程语言的内存模型 [81] 》指出了许多编程语言背后的“万物皆为 X”隐喻; *Smalltalk* 的设计原则 [42] 记录了 Smalltalk 设计中采用的哲学目标与准则; “四人帮”的设计模式 [29]汇编了具体的实现策略; 而符号表示的认知维度 [34] 则为软件的符号表示表层及其权衡提供了一套通用词汇。

后两者直接影响了我们的提案。首先，认知维度是一组可用于分析符号表示的定性属性。我们将这一方法扩展到系统中符号表示之外的“其余部分”，并提出技术 维度。

[2]ACM 用户界面软件与技术研讨会。[3]IEEE 可视化语言与以人为中心的计算研讨会。[4]编程体验。

其次，我们的各个维度自然归属于更大的集群，并以一种常规格式呈现，类似于经典设计模式的呈现方式。对于他人提出的特征，我们的贡献之一是将它们整合到一个共同框架之下：现有的活性、多元主义和统一性隐喻（“一切都是 X”）概念，都成为我们框架中的维度。

方法论。 我们遵循评估编程系统 [18] 的立场，将我们的工作与人机交互方法和实证评估区分开来。我们通常关注那些显然不适用于统计分析（例如挖掘软件代码库）或受控用户研究等实验方法的特征，因此一般不会涉及数值。

以用户界面为重点的人机交互研究似乎也在经历类似的发展。UIST 指南 [55] 要求作者全面评估系统贡献，而该领域也为此类评估制定了启发式方法，例如评估用户界面系统研究 [68]。我们的维度集合为识别编程系统中有趣的方面提供了类似的启发式方法，不过这些维度更关注底层技术属性，而非表层接口。

最后，我们认为，前文所述从编程系统转向编程语言的范式转变掩盖了许多值得重新发掘并进一步发展的编程理念 [74]。因此，我们的方法与 Chang 在科学史与科学哲学领域提出的互补科学理念 [9] 相关。Chang 认为，即使在物理学这样的学科中，已被取代或证伪的理论仍可能包含值得记录的有趣思想。在编程领域，过去的系统会因实证失败之外的诸多原因而遭到弃用，因此 Chang 的互补科学方法尤其适用。

编程系统也需要理论！ 简言之，虽然编程语言已有理论，但编程系统也需要理论。它应适用于从小规模的语言实现到大规模的操作系统。我们应当能够分析不同系统的共性与独特之处，揭示新的可能性，并以有效的方式借鉴以往的工作。用库恩术语来说， [54], 它应能推动一门“常规科学”的发展：逐步填充可能系统空间的地图（图1），从而为未来的设计者构建一个知识库。

3 编程系统

我们通过若干示例系统来介绍编程系统这一概念。这些系统取自三个广义的参考类别：

- 围绕基于文本的编程语言构建的软件生态系统。它们由一系列工具组成，例如编译器、调试器和性能分析器。这些工具可以作为独立的命令行程序存在，也可以集成在集成开发环境中。

- 这类系统类似于操作系统（OS），能够构建执行环境并涵盖整台机器（物理或虚拟）的资源。它们提供通用接口，用于用户与计算机之间以及程序彼此之间的通信。可编程应用程序通常针对特定领域进行优化，提供有限的可编程性，并可在新版本中得到增强。
-

需要指出的是，我们选择这些系统并非意在面面俱到；过去和现在有许多系统我们并不了解，或知之甚少，而且显然无法涵盖尚未创建的编程系统。说明这一点后，我们将继续介绍上述分类中的一些系统。这将帮助读者直观理解编程系统这一概念，并为本文后续内容提供一组可随时参照的示例。

3.1 以语言为核心的系统

基于文本的编程语言存在于边界未明确定义的编程系统中。要讨论编程系统，我们至少需要考虑一种语言，以及一个编辑器和一个编译器或解释器。然而，系统的确切边界取决于设计选择，而这会显著影响我们的分析。

Java 与 Eclipse 生态系统。Java [32] 本身不能被视为一个编程系统，但如果将其置于一个工具生态系统中，它便是一个编程系统。界定系统边界的方式有多种，因此会产生不同的分析结果。一个极简的编程系统由用于编写 Java 代码的文本编辑器和命令行编译器组成。更贴近实际的系统，则是嵌入 Eclipse 集成开发环境 [14]中的 Java。编程系统视角让我们能够看到文本代码之外的一切。以 Eclipse 为例，这包括调试器、重构工具、测试和建模工具、图形用户界面设计器等。这种界定方式得出的编程系统功能强大、使用便捷，但包含大量概念和次级符号。

Haskell工具生态系统。 Haskell 是一个更加专注于语言的编程系统。它通过命令行编译器 *GHC* [59]和 *GHCi* 交互式解释器来使用，并搭配具备语法高亮和自动补全等功能的文本编辑器。任何支持语言服务器协议 [63] 的编辑器都足以构成完整的编程系统。

Haskell 以数学为根基，理解其许多概念时依赖数学直觉。这一背景也体现在它使用的符号表示法中。除了用于编写代码的具体语言语法外，其生态系统还使用非正式的数学符号表示法来描述 Haskell（例如，在学术论文或白板上）。这为操作 Haskell 程序提供了额外的工具。在纸面上进行实验，能够带来一种快速反馈，而其他系统可能需要通过实时编程才能提供这种反馈。

从交互式解释器到笔记本。围绕某种编程语言发展起来的另一种开发者生态系统是 Jupyter 笔记本平台 [51]。在 Jupyter 中，数据科学家将脚本划分为笔记本单元格，交互式地执行这些单元格，并直接在笔记本中查看生成的数据和可视化结果。这将交互式解释器与文学化编程结合起来：交互式解释器可追溯至 20 世纪 60 年代 Lisp 的对话式实现，而文学化编程则于 20 世纪 80 年代末应用于 Mathematica 1。 [52] o [93]

作为一种编程系统，Jupyter 具有许多有趣的特点。编程的主要成果是笔记本本身，而不是一个需要单独编译和运行的应用程序。代码以文档格式呈现，并与其他符号表示交织在一起。代码分成小段编写，每段都能快速执行，因此用户能比使用传统编程方式更快获得反馈。笔记本可以看作一份记录，展示结果是如何得出的；然而，笔记本一个常见且棘手的问题是，有些笔记本允许用户不按顺序运行代码块。代码会操纵在后台运行的“内核”中的可变状态。因此，在笔记本中重走一遍操作步骤，比方说在 Common Lisp 中，要微妙得多；在那里， [85],dribble 函数会直接将用户的会话记录到文件中。

3.2 类操作系统编程系统

“类操作系统”始于20世纪60年代，当时人们开始能够与计算机进行一对一交互。首先，分时系统让用户能够通过电传打字机交互式地共享使用计算机；PDP-1 和 PDP-8 等较小型计算机也提供了类似的直接交互方式，而 Alto 和 Lisp 机器等20世纪70年代的工作站则增添了图形显示器和鼠标输入。这些类操作系统 系统的突出特点是具有操作系统的全面统辖范围，无论人们通常是否认为它们承担了这一角色。

MacLisp 和 Interlisp。LISP 1.5 [60] 出现于交互式计算机兴起之前，但由于有解释器且没有声明，使用 Lisp 进行交互式操作自然而然；最早的此类实现于 20 世纪 60 年代初问世。Lisp 家族的两个分支——MacLisp 和后来出现的 Interlisp——都采用了交互式的“对话式”工作方式，最初借助电传打字机，后来则使用屏幕和键盘。

MacLisp 和 Interlisp 都采用了持久地址空间这一理念。关闭系统电源时，程序代码和程序状态都会保留下来；用户既可以通过程序，也可以通过相同的方式进行交互式访问和修改。Lisp 机器秉持这样的理念：机器持续运行，并在需要时将状态保存到磁盘。如今，云服务（如 Google 文档和在线集成开发环境）中广泛采用了这一理念。MacLisp 和 Interlisp 首创的另一项理念是使用结构体编辑器。借助这些编辑器，程序员可以将 Lisp 数据结构视为嵌套列表，而不是字符序列来操作。在 Interlisp 中，程序员可以使用*P 等命令打印当前表达式，或使用*(2(X Y)) 将其第二个元素替换为参数(XY)。PILOT 系统 [89] 还提供了更先进的对话功能。遇到排版错误和其他失误时，

它会提供自动修正方案，供用户交互式确认；确认后，系统会修改内存中的程序并恢复执行。

Smalltalk。Smalltalk 于20世纪70年代问世，其独特目标是提供“所有年龄段的人都能使用的动态媒体” [43]。作者将计算机视为一种元媒体，能够成为教育、话语、创意艺术、模拟以及其他尚未被发明的应用所需的一系列其他媒体。Smalltalk 是为配备图形显示器的单用户工作站设计的，并率先将这种显示方式不仅用于应用，也用于编程本身。在 Smalltalk 72 中，用户使用由鼠标控制的结构编辑器在屏幕下半部分编写代码，并通过菜单编辑定义。在 Smalltalk-76 及后续版本中，这种方式转变为嵌入类浏览器 的文本编辑，用于浏览类及其方法。

与 Lisp 系统一样，Smalltalk 采用持久地址空间的编程模型，所有对象都保留在内存中，但其基础是对象 和消息传递，而不是列表。通过编程或执行对系统状态所做的任何更改，都会在计算机关闭后保留下来。最后，由于 Smalltalk 环境的大部分内容都是用其自身实现的，因此可以从系统内部对系统进行广泛修改。

我们将 Lisp 和 Smalltalk 归入类操作系统，因为它们在许多方面都发挥着操作系统的作用。在 Xerox Alto 和 Lisp 机器等专用机器上，用户启动机器后会直接进入 Lisp 或 Smalltalk 环境，并能够在系统内部 完成所有需要做的事情。如今，这种体验则与 UNIX 及其衍生系统联系在一起，后者运行于各种各样的通用机器上。

UNIX。UNIX 说明，编程系统的许多方面都受到其预期目标受众的影响。UNIX 为计算机黑客打造， [57], 其抽象和接口都贴近机器。尽管 UNIX 在历史上与 C 语言密切相关，但它发展出一套与语言无关的抽象，使得在同一系统中使用多种编程语言成为可能。在 Smalltalk 中，一切都是对象；而 UNIX 系统的本体则由文件、内存、可执行程序和运行进程构成。请注意这里明确的“阶段”区分：UNIX 区分易失性的内存 结构（系统关闭后便会丢失）与非易失性的磁盘 结构（能够保留）。Lisp 和 Smalltalk 则将这种内存类型之间的区分视为实现细节，并通过持久地址空间将其抽象掉。尽管如此，这并未妨碍 UNIX 的本体支持由不同语言和工具构成的多元生态系统。

早期与现代万维网。万维网 [1] 从一个用于共享和组织信息的系统，演变为一种编程系统。如今，它由种类繁多的服务器端编程工具、JavaScript 及可编译为 JavaScript 的语言，以及 HTML 和 CSS 等表示法构成。作为一种编程系统，“2020年代现代 Web”与“20世纪90年代初期的 Web”有明显区别。在早期万维网中，JavaScript 代码以便于复制和重复使用现有脚本的形式分发，这促使非专业人士热情地

采纳它——这让人想起早十年间，Commodore64 搭载 BASIC 的微型计算机诞生时的情景。

在“现代万维网”中，多种编程语言都将 JavaScript 作为编译目标，而 JavaScript 也被用作服务器端语言。如今的万维网已不再简单到足以鼓励人们从不同网站复制粘贴代码并加以改编。不过，它确实配备了先进的开发者工具，提供了类似早期 Lisp 和 Smalltalk 交互式编程系统的功能。网页创建的文档对象模型（*DOM*）结构是透明的，用户可以访问它，并通过内置的浏览器检查器工具进行修改。第三方代码可以通过扩展程序注入，以修改 DOM。DOM 几乎类似于 Smalltalk 和 Lisp 映像中的树／图模型，只是缺少持久化这一关键属性。不过，Webstrates [50] 正在解决这一局限。

3.3 面向应用的系统

前文讨论的编程系统，要么是通用系统，不专注于任何特定类型的应用；要么面向广泛的领域，例如 Lisp 中的人工智能和符号数据操作。相比之下，以下示例聚焦于需要构建的、更为具体的应用类型。其中许多系统支持基于丰富交互以及专用视觉和文本符号表示进行编程。

电子表格。第一批电子表格于1979年问世，名为VisiCalc [33, 94]，帮助分析师进行预算计算。作为编程系统，电子表格的显著特点是其编程基底（二维网格）和求值模型（自动重新求值）。随着时间推移，电子表格的可编程性不断发展，逐渐增加了各种功能，使其成为强大的编程系统，而VisiCalc并非如此。最后一步是1993年在Excel中加入宏，之后又通过*Visual Basic for Applications*进一步扩展。

图形化“语言”。不依赖文本代码来支持编程的尝试，只能在比喻意义上称为“语言”。在这些编程系统中，程序由图形结构构成，例如LabView [53] 或示例编程 [58]。

HyperCard。电子表格旨在解决特定应用领域中的问题，而HyperCard [62] 则围绕一种特定的应用格式设计。程序是包含多媒体组件和按钮等控件的“卡片堆栈”。这些控件可以通过“导航到另一张卡片”等预定义操作进行编程；要实现更复杂的功能，则可使用HyperTalk脚本语言。

作为一种编程系统，HyperCard 有几个值得关注之处。它有效地结合了视觉符号表示和文本符号表示。程序在编辑时和执行时呈现的形式相同。最值得注意的是，HyperCard 支持用户逐步转变为开发者：用户可以先使用堆栈，然后

继续编辑视觉元素或选择预定义逻辑，直到最终学会使用 HyperTalk 编程。

4 技术维度目录

我们将在此详细介绍所提出的技术维度。请注意，我们的意图是提供一份可供查阅并按需使用的参考资料，而不是一份应从头读到尾的材料。因此，在继续阅读第 5 节之前，我们建议先快速浏览本文，了解其中是否有特别感兴趣的内容。在第 5 节中，我们将结合具体示例介绍若干维度；届时，返回本文查阅更多细节可能会有所帮助。为便于快速了解整体内容，我们提供了摘要表 1、2 和 3，不过在阅读相关章节后，这些表可能会更容易理解。

我们将这些维度按集群分组。在研究给定系统时，可以将集群视为“感兴趣的主题”或“探究领域”，将相关维度归为一组，用于对该系统进行衡量。

每个集群均有名称，并以一个加框的摘要开头，随后是较长的讨论，最后列出与其他集群的任何关系，以及适用时的参考文献。在主体讨论中，会列出各个维度。有时，某一维度上的特定取值（或多个维度上多个取值的组合）可以被识别为一种熟悉的模式——或许在相关文献中已经有既定名称。这些内容会标记为示例。最后，穿插在讨论中的、既不是维度也不是示例的内容，将以备注的形式介绍。

我们再次强调，我们并不认为所提出的目录能够涵盖编程系统过去和未来的所有方面。我们欢迎后续研究对我们在此着重探讨的方面加以拓展或质疑。

4.1 交互

用户如何表达自己的想法、评估结果，并据此产生新的想法?

编程系统的一个重要方面，是用户在创建程序时如何与系统交互。以采用静态类型、编译型语言且库链接方式直接明了的标准形式为例：程序员在文本编辑器中编写代码，调用编译器，然后阅读收到的错误消息。修复代码并通过编译后，遇到运行时错误时也可能会经历类似的过程。

其他形式也同样可能存在。一方面，某些常见交互（例如编译或执行程序）可能完全不可察觉。另一方面，系统可能会提供各种接口，以支持编程中往往十分重要的诸多其他交互，例如查阅文档、管理依赖项、重构或结对编程。

表 1 技术维度速查表。

维度（集群）摘要		关键示例范围
交互	用户如何表达他们的想法 并据此产生新的想法?	，评估结果 ，
反馈循环	各种.....有多宽 执行的 执行 和 求值- *uation* 以及它们是如何重新- 了吗?	即时 反馈 （短）与 批处理 mode （长）评估鸿沟
交互模式	哪些反馈集合 循环只会共同出现?	设置 vs. 编辑 vs. 调试- ging
抽象 构造	我们如何从抽象概念 转向具体示例 反之又如何?	示例编程 与第一性原理
符号表示	不同的文本／可视化编程符号表示 法之间有何关联?	
符号结构	使用了哪些表示法 来编写系统程序，以及 它们之间有何关联?	表示法重叠，需要 同步与互补彼此 其他
表层／内部 表示法	它们之间的联系是什么—— 用户看到的内容与 计算机程序看到的内容 之间有何区别?	序列编辑与渲 染、结构编辑与 恢复
主要 / 次要 表示法	是否有某种表示法比其他表示法更重 要?	辅助构建脚本与 可视化编辑器和代码 在 Flash 中处于平等地位
表达式分布	相似表达式是否能编写 相似的程序?	简洁但容易出错，相较于 明确但冗长
表示法的一致性 表示法	该符号表示使用的是 少量还是大量的 基本概念?	Lisp S-表达式 vs. 类似英语的文本注 释

■ **表 2** 技术维度速查表（续）。

维度（集群）	摘要	关键示例范围
概念结构	意义是如何构建的？内部和外部激励如何取得平衡？	
概念完整性与开放性	这个系统呈现出优雅的设计，还是务实地临时拼凑出来的？	完整性（一切皆为X）与开放性（兼容混合）
可组合性	有哪些基本元素？如何将它们组合起来以实现新颖的行为？	顺序、选择、重复、函数抽象，递归，逻辑连接词
便利性	用户不必重复发明哪些轮子？	小型还是庞大的标准库
共性	有多少< शैली id='1'>通用结构被明确标记为这样吗？	共同结构被冗余展开，而非因式分解out
可定制性	程序存在于系统中后，如何对其进行扩展和修改？	
定制分阶段定制	我们必须对运行中的程序进行不同的定制吗？与静态程序相比吗？这些更改会持续到终止之后吗？	源代码与配置文件，开发者工具选项卡，自动基于图像的持久化，脚本语言
可外部化性 / 增量式创作	哪些部分of the系统的状态可以被引用and转移向它传入或从中获取？可以在多大程度上the系统的行为被改变by通过添加表达式？	状态是：(*i*) 私有且隐藏；(*ii*) 可外部化且可覆盖；(*iii*) 外部化且可覆盖，通过添加-tion
自我可持续性	系统的能力有多强行为可以从内部改变？	无（依赖外部工具）与自给自足（包含所需的一切）

■ **表 3** 技术维度速查表（续）。

维度（集群）	摘要	关键示例范围
复杂性	系统如何组织复杂性，以及需要达到什么程度的细节?	
复杂性分解	需要哪些编程细节隐藏在可复用的组件中以及如何隐藏?	领域特定（更多隐藏）与通用（更少隐藏）
自动化程度	程序逻辑的哪一部分不需要明确指定? 不需要明确指定?	垃圾回收（低技术）vs. Prolog 引擎（高科技）
错误	系统认为何为错误？如何防止并处理这些错误?	
错误检测	可以检测到哪些错误 在哪些反馈循环中， 以及如何检测?	实时编码中的人工检查 与部分自动化相比 静态类型检查中的作用
错误响应	系统如何在检测到错误时作出响应?	它会停止、自动恢复地，忽略错误 或询问用户如何继续?
可采纳性	系统如何促进或阻碍个人和社区采纳?	
易学性	对……的态度如何? 学习曲线如何？目标受众是什么?	面向普通公众的 HyperCard 与面向科学界的 FORTRAN 科学家
社交性	哪些社会和经济因素塑造了这个系统的现状?	大教堂与集市模式

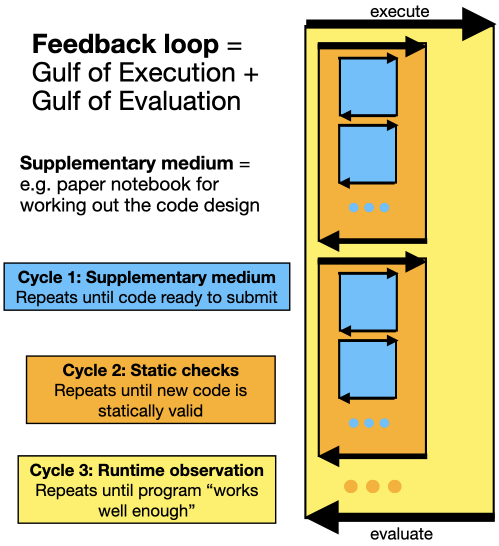


图 2 静态检查编程语言中的嵌套反馈循环。

我们关注程序员与系统交互、构建具有预期行为的程序时所发生的交互。为了分析这些交互，我们借用执行鸿沟和评估鸿沟 这两个概念，它们来自《日常事物的设计》[67]。

4.1.1 维度：反馈循环

使用系统时，人们首先会有某种想法，并尝试让它在软件中实现；用户目标与实现该目标的手段之间的差距称为执行鸿沟。然后，人们会将实际取得的结果与心中的原始目标进行比较；这一过程跨越了评估鸿沟。这两项活动构成了反馈循环，用户通过它逐渐实现想象中的愿望，或不断完善这些愿望，以弄清“自己真正想要什么”。

一个系统必须至少包含一个这样的反馈循环，但也可以在不同层级上包含多个反馈循环，或包含专用于特定领域的反馈循环。对于每个反馈循环，我们都可以将执行鸿沟和评估鸿沟视为旅程中彼此独立的阶段，而跨越它们的方式和速度可能各不相同。

例如，我们可以将经过静态检查的编程语言 （如 Java、Haskell）分析为几个反馈循环（图 2）：

1. 程序员在编写代码之前，常常会在白板或笔记本上思考设计细节并进行计算。这种辅助媒介 也有自己的反馈循环，尽管这一过程通常不是自动的。

2. 编写代码后，会将其交由静态检查器检查。出现错误时，用户会返回编写代码；如果检查成功，他们就“获准”运行程序，从而进入循环 3。

- 执行鸿沟包括补充媒介的多个周期，以及调用编译器（例如通过构建系统）所需的任何开销。

- 评估鸿沟本质上是观察到静态错误或成功终止之前的等待时间。因此，它受代码长度的某个函数所限制（对于接下来的周期 3，则不能这样说。）

3. 有了可运行程序后，用户现在可以评估其运行时行为。运行时错误可能会使用户回到编写待检查代码的阶段，也可能让用户在类似的循环中调整动态加载的数据文件。

- 这里的执行鸿沟可能包含多次循环 2 的迭代，每次迭代又各自嵌套一个循环 1。这里的评估鸿沟在理论上没有上限；人们可能需要等待很长时间，或者创
- 造非常具体的条件来排除某些错误（例如竞态条件），或者仅仅是为了判断该程序是否适合其用途。通过施加静态检查，可以将某些错误提前到循环 2 的评估阶段，从而缩小循环 3 的评估鸿沟可能达到的大小。另一方面，这可能
- 会使编写通过静态检查的代码变得更困难，从而增加第 2 级循环的次数，进而扩大第 3 级的总执行鸿沟。根据两者的平衡情况，顶层反馈循环的总时长
- 可能变长，也可能变短。

-

4.1.2 示例：即时反馈

将<style id='1'>评估鸿沟缩小到无法察觉的具体情形，被称为即时反馈。一旦用户对系统进行了某种更改，其影响（包括错误）就会立即显现。这是活性的关键要素，尽管其本身并不足够。（参见关系）

实现即时反馈的便捷性显然受到用户操作对系统造成的计算负载，以及系统执行此类任务时的性能所限制。不过，这种“加载时间”并不是反馈延迟的唯一原因：一种常见情况是，用户在执行操作后必须手动请求（或“轮询”）系统的相关状态，即使系统已经迅速完成了任务。在这种情况下，反馈可以描述为按需即时提供，但并非自动请求。为方便起见，我们选择将后一个标准——自动请求结果——纳入即时反馈的定义。

在 交互式解释器 或命令行 *shell*中，存在一个主要 循环：输入命令并查看其输出；也存在一个次要 循环：输入命令行本身并检查它。命令的输出可能会立即显示，但通常只反映全部效果的一部分，甚至毫无反映。用户必须随后手动发出更多命令，逐步检查相关状态。次要循环和所有输入操作一样，会通过字符“回显”提供即时反馈，但语法错误之类的问题通常要等到整行提交之后才会报告。这种评估鸿沟在网络浏览器的 JavaScript 控制台中有所缩小，用户每敲一个键，当前行都会以有限的方式“运行”。

每次按键时都会运行。没有副作用的简单命令，[5] 例如调用纯函数，可以立即显示预览结果——不过，尚未输入完整的表达式和语法错误不会触发预览。

4.1.3 示例：直接操控

直接操作 [80] 是即时反馈循环的一种特殊情况。用户以尽可能接近现实生活的方式查看并与人工制品交互；这通常包括用光标或手指拖动，以实际移动视觉项目，而具体操作方式受所用触觉技术的限制。

自然，用手移动真实物体时无需等待对象“跟上”，因此[6] 直接操作必然是一个即时反馈循环。另一方面，如果要通过在文本框中输入新的坐标来移动屏幕上的图形，那么这仍可能提供即时反馈 （如果更新即时且自动出现），但并不 属于直接操作的范例。

电子表格 包含一个用于直接操作数值和格式的反馈循环，就像任何其他所见即所得应用程序一样。在这里，每输入一个字符、每更改一种样式，都会得到反馈。公式编辑和公式调用则属于另一个循环，情况并非如此。在那个循环中，设计和输入公式时的执行鸿沟更大，只有按下回车键提交公式后才会给出反馈。因此，按照上文的定义，这只是一个按需提供“即时反馈”的循环。

4.1.4 维度：交互模式

编程系统中的交互通常会以特定的模式组织起来，使交互及其相关的反馈循环只在某些模式下可用。例如，创建新项目时，用户可能可以通过对话式接口配置项目，例如现代 JavaScript 中的npminit 。项目创建后，这类交互便不再可用。交互模式这一理念并不局限于编程系统，也出现在软件工程方法论中。具体来说，将实现阶段与维护阶段分开，就是两种模式的一个例子。

编辑与调试。 一个很好的例子是编辑 模式与调试 模式之间的区别。调试程序时，用户可以修改程序状态，并立即（或更及时地）获得有关各项运行操作的反馈。在某些系统中，甚至可以在调试过程中修改程序本身。调试模式之外无法实现这样的反馈循环。

[5]当然，这些都是通过某种保守的过近似检测出来的，这种过近似会排除那些可能产生副作用的表达式。[6]在某些情况下，例如用船舵操纵船只，输入与效果之间会有延迟。但仔细观察就会发现，延迟发生在船舵与船之间；我们不会看到手像全息影像一样穿过方向盘，然后方向盘在一秒后才转动。在现实生活中，直接触碰的对象会立即给出反馈；而控制链条更远端的对象则未必如此！

Lisp 系统 有时会区分解释 模式和编译 模式。这两种模式的区别不仅在于代码执行效率，还在于它们支持的交互方式。解释模式下，可以交互式地测试代码，并在代码执行过程中修正错误（参见错误响应）。编译模式下，只能对整个程序进行测试。例如，一些 Haskell 系统也有这两种模式，其中 REPL 使用的解释器（GHCi）与编译器（GHC）相互独立。

Jupyter 笔记本。编程系统也可以统一通常彼此独立的模式。Jupyter 笔记本环境没有独立的调试模式；用户运行代码块并获得结果。这种单一模式可用于快速试验，也可用于生成最终结果，在一定程度上兼具调试模式和编辑模式的作用。不过，即使是 Jupyter 笔记本，也区分编辑文档和运行代码。

4.1.5 维度：抽象构造

编程中一项必不可少的活动，是在抽象模式与具体实例之间转换。抽象可以根据具体示例、第一性原理或通过其他方法构造。这个过程的一部分可能发生在程序员的头脑中：他们思考具体案例，提出一个抽象概念，然后将其直接编码到系统中。或者，系统也可以直接支持这些不同的方法。

一种选择是从第一性原理构造抽象。在这种情况下，程序员首先定义一个抽象实体，例如面向对象编程语言中的接口。为此，他们必须先（在头脑中）思考所需的抽象，然后将其编码（到系统中）。

另一种选择是从具体案例构建抽象。在这种情况下，程序员会使用系统解决一个或多个具体问题；当他们对此感到满意后，系统便会引导他们基于具体案例创建抽象。在编程语言集成开发环境中，这体现为“提取函数”重构；而在其他系统中，我们则可以看到宏录制等方法。

Pygmalion。 在 Pygmalion [84], 中，所有编程都是通过操作代表具体事物的具体图标来完成的。要创建抽象，可以使用“记忆模式”，它会记录对图标执行的运行，并使你能够将这段记录绑定到一个新图标上。

Jupyter 笔记本。 在 Jupyter 笔记本中，你往往会直接处理具体事物，因为每个单元格之后都会显示预览。这会让人不愿创建抽象，因为这样一来，你就无法再以如此精细的粒度查看内部内容。

电子表格。 直到最近 Excel 引入 λ 表达式之前，电子表格一直都非常具体，除了复制粘贴之外，没有其他方式可以抽象并重用计算模式。

4.1.6 关系

- 错误（第4.6节）更长的评估鸿沟会延迟错误的发现。更长的执行鸿沟会增加发生错误的可能性（例如，编写大量代码或

花很长时间编写代码）。将运行时错误转变为静态检测到的错误，可以缩短评估鸿沟与执行鸿沟的总和。

- 可采纳性 （第4.7节）：执行 鸿沟关乎软件的使用和一般意义上的编程。将一个想法转化为软件所需的时间，受用户的熟悉程度和系统的易学性影响。

- 符号表示 （第4.2节）：反馈循环与符号结构相关。在包含多种表示法的系统中，每种表示法可能对应不同的反馈循环。在实时编程的语境中，“屏幕上的东西应该就是实际的东西”这一格言 [71], 与活性 有关，体现了表层与内部符号表示之间的直接联系。其理念是，可交互对象应具有忠实的行为，而不应只是隐藏的真实 对象投下的无形影子。

4.2 符号表示

不同的文本／视觉编程符号表示法之间有何关联?

编程总是借助某种形式的符号表示来完成。我们以最广义的方式看待表示法，并将使用文本或视觉符号表示的任何结构化手势都纳入其中。文本符号表示主要包括编程语言，也包括配置文件等。视觉符号表示包括图形化编程语言。其他类型的结构化手势还包括用于构建系统中视觉元素的用户界面。

4.2.1 维度：符号结构

在实践中，大多数编程系统会使用多种表示法。不同的表示法在系统中可以发挥不同的作用。一方面，系统可以提供多种重叠符号表示，让用户能够以不同方式对系统的相同方面进行编程。在这种情况下，每种符号表示可能更适合不同类型的用户，但也可能存在某些局限（例如，视觉符号表示的表达能力可能有限）。另一方面，也可以使用多种互补记法 来对系统的不同方面进行编程。在这种情况下，对系统进行编程需要使用多种表示法，但每种表示法可能更适合当前任务；例如，HTML 描述文档结构，而 JavaScript 则指定其行为。

4.2.2 示例：重叠符号表示

编程系统可能会为编写系统的同一方面提供多种表示法。这样做通常是为了便于完成不同任务：例如，用文本符号表示来定义抽象，用视觉符号表示来指定具体结构。此类安排中的关键问题是同步不同的表示法；如果它们具有不同的特性，这种映射可能并不简单。例如，源代码可能支持更复杂的抽象机制，如循环，而这些循环在视觉符号表示中会呈现为可见的重复。当用户编辑一个由此类重复产生的单个对象时，系统应该如何处理？类似地，文本符号表示可能

允许不完整的表达式，而这些表达式在视觉符号表示中没有对应形式。对于使用重叠符号表示的编程系统，我们需要说明这些表示法如何同步。

Sketch-n-Sketch [38] 使用重叠符号表示来创建和编辑 SVG 与 HTML 文档。用户在一个分屏结构的界面中编辑文档，左侧显示源代码，右侧显示渲染后的视觉输出。用户可以同时编辑两者，所做的更改会同步到另一个视图。代码可以使用抽象机制（例如函数），而这些机制无法在可视化编辑器中完全显示（我们会在下文的表达式分布 中再次讨论这个问题）。Sketch-n-Sketch 可视为投影式编辑器的一个示例。[7]

UML 往返转换。 另一个采用重叠符号表示 结构的编程系统示例，是 UML 设计工具，它们同时以源代码和 UML 图的形式显示程序。在一种表示法中进行的编辑会自动更新另一种表示法。Together/J[8] 系统就是一个例子。为了解决符号表示同步的问题，这类系统通常需要在文本符号表示中存储额外信息，一般使用一种特殊的代码注释。在这个例子中，用户重新排列 UML 图中的类之后，需要在代码中更新这些新位置。

4.2.3 示例：互补记法

编程系统还可以为其领域的不同方面提供多种互补记法，以便对这些方面进行编程。同样，这通常是为了让某些方面的指定更加容易；但当不同方面能够更加清晰地分离时，这种做法更为合适。采用互补记法的系统，其关键问题在于不同记法之间如何关联。用户可能需要同时使用两种记法，也可能需要在解决日益复杂的问题时，逐步从一种记法过渡到下一层级。在后一种情况下，如何易学地从一个层级过渡到下一个层级，是一个主要考量。

*电子表格和 HyperCard。*在 Excel 中，有三种互补记法，允许用户指定复杂程度逐步提高的内容：(i) 可视化网格、(ii) 公式语言，以及 (iii) Visual Basic for Applications 等宏语言。这些表示法在很大程度上相互独立，表达能力也各不相同。在网格中输入数值无法用于指定新的计算，但可以用来调整或运行计算，例如在假设分析中输入不同的备选方案。更复杂的任务则可以通过公式和宏来完成。用户会逐步学习更高级的表示法，但此前使用某种表示法的经验并不能帮助他们掌握下一种表示法。这种方法针对某一层级的易学性进行了优化，却也给用户设置了一道必须跨越的门槛，才能进入第二层级。*HyperCard* 的符号结构与此类似，并且

[7]从技术上讲，传统的投影式编辑器通常会更直接地处理编程语言的抽象语法树。[8] https://www.mindprod.com/jgloss/togetherj.html，访问日期：2023-02-05。

由以下部分组成：(i) 卡片的可视化设计、(ii) 通过图形用户界面进行的可视化编程（可执行的运行种类有限），以及 (iii) 用于任意脚本编写的 HyperTalk。

Boxer 和 *Jupyter*。 Boxer [16] 采用了互补记法，将视觉符号表示（文档的布局及其由之构成的方框）与文本符号表示（方框中的代码）结合起来。在这里，文本符号表示始终嵌套在视觉符号表示之中。Jupyter 笔记本的情况也类似。文档结构是图形化的；代码和视觉输出以可编辑单元格的形式嵌套在文档中。这种安排在许多其他系统中也很常见，例如 Flash 或 Visual Basic，它们都将视觉符号表示与文本代码结合起来，不过两者并非相互嵌套。

4.2.4 维度：表层符号表示与内部符号表示

所有编程系统都会在内存中构建结构体，我们可以将其视为一种内部符号表示，通常用户无法看到。尽管这类结构体可能会在调试器中显示出来，但在正常运行期间它们是隐藏的。用户实际交互的是表层符号表示，通常是屏幕上的文本或图形。每次与表层符号表示交互，都会以某种方式改变内部符号表示；这种联系的性质值得进一步探讨。为此，我们用一个简化的二元选择来说明这些符号表示的形式。

4.2.5 示例：隐式结构与显式结构

我们将表示法分为两类。具有隐式结构 的表示法以项目序列的形式呈现，例如文本字符或音频信号振幅。具有显式结构 的表示法则呈现为树或图，而没有明显的顺序，例如矢量图形编辑器中的形状。这两类表示法可以相互转换：字符串中包含的隐式结构可以解析 为显式语法树，而显式文档结构则可以渲染 为具有相同隐式结构的字符序列。

现在，考虑一个用于输入个人姓名的接口，姓名由名字和姓氏组成。对于表层符号表示，可以用一个文本字段存放姓名，并以空格分隔；此时，子结构隐含在字符串中。或者，[9]也可以设置两个字段，分别输入名字和姓氏，从而显式表示二者的分隔。内存中构建的内部符号表示也面临类似的选择：它是一个字符串，还是两个独立的字符串？

我们可以看到，这些选择形成了四种组合。更有意思的是，由于两个关键的不对称性，这些组合各自具有独特的特点。首先，表层符号表示主要供人类使用，而内部符号表示主要供计算机使用。其次，也是最重要的一点，计算机程序只能处理显式结构，而人类既能理解显式结构，也能理解隐式结构。鉴于这种不对称性会带来实际影响，我们将先重点考察包含内部符号表示的组合。

[9]关于可选方案令人惊讶的多样性，请参见《单一事实的多种形式》 [47] 。 e.

4.2.6 示例：内存中的一个字符串（隐式结构化内部符号表示）

最简单的情况是，表层符号表示中采用隐式结构，即用一个文本框显示完整姓名。对表层内容所做的编辑会直接反映到内部，并保存到磁盘上。这对应于文本编辑。如果把基本操作视为随时间将事件记录到列表中，我们就可以将这一概念推广为序列编辑。对于文本，这些事件是按键；对于音频编辑界面，它们则是声音振幅采样。

在另一种情况下，使用两个文本框时，我们有序列渲染。接口中有关两个字符串分隔的信息并没有完全被“丢弃”，而是作为字符串中的一个空格字符变得隐式。这种组合对应于 Visual Basic 从图形用户界面表单生成代码、视频编辑器将多个片段和效果合并为单个流，以及 3D 渲染器将场景图转换为像素。另一个例子是基于行的差异工具，它们提供并排视图和相关接口，但最终必须将用户的更改传递给底层文本文件。

关键在于，在这两种情况下，计算机程序只能将存储的顺序作为顺序来处理；也就是说，只能执行插入、删除或按顺序读取。这里吸引人的特点是，这些运行易于实现，并且可以在多种类型的顺序之间复用。然而，任何进一步的结构都是隐式的；若要以编程方式处理它，用户必须编写程序将其解析为某种显式结构。此外，在此阶段引入的错误可能只是被记录到顺序中，直到很久以后尝试使用数据时才被发现。

4.2.7 示例：内存中的两个字符串（显式结构化的内部符号表示）

有两个文本框时，两种表示法相互匹配，因此无需做太多工作。与序列编辑一样，可以将对表层的编辑同步到内部符号表示。这对应于矢量图形编辑器和 3D 建模工具，以及面向编程语言的结构编辑器 。因此，我们将这种组合称为结构编辑。

只有一个文本字段时，我们采用结构恢复。 每次文本发生变化时，都需要进行解析。这种方式常见于浏览器开发者工具中的 DOM 检查器，在那里可以将 HTML 作为文本编辑，以更改文档树结构。更一般地说，编译器和解释器也采用这种模式：它们接受程序源文本，但内部处理的是树和图结构。通过这种方式，也可以进行某种结构编辑：让操作体验类似文本编辑，但输出内容具有明确的结构。

在这两种情况下，为了编写程序来转换、分析或以其他方式处理用户创建的数字制品，可以直接遍历已存储的结构，而无需每次使用时都对其进行解析。解析要么被完全省略，要么被简化为编辑过程中发生的短暂过程；这意味着可以在错误被引入的那一刻捕获它们，而不是让它们一直潜伏。

4.2.8 维度：主要表示法与次级符号

实际上，大多数编程系统都会使用多种表示法。即使是基于传统编程语言的系统，该语言的主要表示法通常也会由次要表示法提供支持，例如编码在注释和构建工具配置文件中的注解。不过，多种表示法也可以同时作为主要表示法，尤其是在它们如前文所定义的那样相互重叠时。

编程语言。 围绕传统编程语言构建的编程系统通常还拥有与之相关的其他表示法或结构化手势。UNIX 中的主要表示法是 C 编程语言。然而，它被置于一个提供多步骤机制的编程系统中，使程序员能够通过终端运行 C 代码，并由 Shell 脚本等次级符号辅助。一些编程系统试图将通常依赖次级符号的工具集成到系统本身中，从而减少程序员需要掌握的次级符号数量。例如，在 Smalltalk 的后代 Pharo 中，版本控制和包管理都在 Pharo 内部完成，因此无需使用git 命令和依赖配置文件等次级符号。[10]

Haskell。 在 Haskell 中，主要表示法是编程语言，但同时也存在许多次级符号。其中包括包管理器（例如cabal.project 文件）或 Haskell 构建工具的配置文件。更有趣的是，Haskell 还关联着一种非正式的数学符号表示法，程序员在白板上讨论程序或在学术出版物中撰写内容时会使用这种表示法。采用这种数学符号表示法的想法可以追溯到*Algol* 报告 *58* [72]„ 该报告明确规定了一种“出版语言”，用于使用希腊字母和下标“陈述和交流问题”。

4.2.9 维度：表达式地理

一种符号表示的关键特征，是其结构与其所编码的行为结构之间的关系。最重要的是，在特定符号表示中，相似的表达式是否代表相似的行为？[11] 视觉符号表示可能提供或多或少直接的映射。一方面，积木式语言中外观相似的代码可能意味着截然不同的事物。另一方面，两张 HyperCard 卡片的设计如果看起来相似，最终呈现出的卡片也会相似——符号表示与逻辑之间的映射要直接得多。

*C/C++ 表达式语言。*在文本符号表示中，情况可能并非如此。请看下面两个 C 条件语句：

- `if (x==1) {... }` 计算布尔表达式 `x==1`，以确定 `x` 是否等于 `1`，如果条件成立则运行代码块。

[10]Pharo 中用于版本控制和包管理的工具，仍可视为一种内部领域特定语言，因而也是一种次级符号表示；不过，它的基本结构与 Pharo 系统中的其他符号表示共享。[11]另请参阅 Basman 对“密度”的 [2] 类似讨论。

- if (x=1){ ... } 将 1 赋给变量 x。在 C语言中，赋值是一个返回所赋值的表达式，因此结果 1 会被解释为真，代码块也始终 会执行。

可以设计一种符号表示，使其更好地对应其背后的逻辑，例如，要求用户写入 1== x。这样就解决了上述问题，因为 1 是字面量而非变量，因此不能对其赋值（1=x 会导致编译错误）。

4.2.10 维度：表示法的一致性

表示法的一个常见关注点是其一致性程度。一致的表示法只用少量概念，就能表达种类繁多的事物。这里的主要例子是 Lisp 的 S-表达式。S-表达式要么是一个原子，要么是一对 S-表达式，写作 (s1. s2)。按照惯例，S-表达式 (s1 . (s2 . (s3. nil))) 表示一个列表，写作 (s1s2 s3)。在 Lisp 中，表示法的一致性与表示的一致性密切相关。[12] 在 LISP 1.5 的理想化模型中，S-表达式所表示的数据结构就是内存中实际存在的内容。在现实世界的 Lisp 系统中，内存中的表示则更为复杂。编程系统也可以采取截然不同的方法，将符号表示与内存中的表示完全分离。

Lisp 系统。 在 Lisp 中，源代码在内存中以 S-表达式的形式表示，可以通过 Lisp 原语对其进行操作。此外，Lisp 系统将强大的宏处理纳入其语义之中：展开宏会修改使用该宏的代码的列表结构。二者结合，使得我们可以用 Lisp 为系统定义扩展，并使其语法与 Lisp 本身无法区分。此外，还可以编写一个程序来构造另一个 Lisp 程序，并且不仅能以解释方式运行它（使用 eval 函数），还能在运行时编译它（使用 compile 函数）并执行它。许多领域特定语言以及新编程语言（例如 Scheme）的原型，都是通过这种方式实现的。从这个意义上说，Lisp 这门语言是一种“可编程编程语言”。 [21, 19]

4.2.11 参考文献

符号表示的认知维度 [34] 为分析单个符号表示提供了一个全面的框架，而我们在此关注的是多种符号表示之间的关联及其结构。值得注意的是，符号表示的认知维度也定义了次级符号，但其含义与我们所说的不同。对他们而言，次级符号指的是一种符号表示是否允许加入颜色或注释等冗余信息，以提高可读性。

更广泛而言， [49] 曾将表示法在科学实践中的重要性研究为“纸上工具”。这些类似公式的实体可由人类进行操作，以替代实验，例如前文所述的 Haskell 数学符号表示法：一种用于在白板上进行实验的“纸上工具”。程序

[12]表示法通常与表示密切相关，因为表示法可能会反映用于程序表示的结构体。Basman 等人 [4] 将此称为“无生命的”表示法与“有生命的”表示形式之间的区别。

设计表示法与之类似，但它们是与机器进行通信的一种方式；实验并不只发生在纸面上。

4.2.12 关系

- 交互 （第4.1节）：编程系统中存在的反馈循环通常与各个符号表示相关联。不同的符号表示也可能具有不同的反馈循环。可采纳性 （第4.7节）：符号结构会影响易学性。特别是，互补记法可能要求多个（且可能不同的）用户掌握多种符号表示。重叠符号表示则可能提升易学性，因为用户可以用一种方式（或许是可视化方式）编辑程序，并在另一种表示法（例如代码）中看到效果。错误 （第4.6节）。仅按顺序记录用户操作（例如文本编辑）的过程，尤其会记录用户犯下的任何错误，并将其留待之后使用数据时处理，使错误保持潜伏状态。相反，将用户操作视为对结构的编辑，并施加约束和正确性规则的过程，则能够在错误出现时立即发现它们，并确保输出的数据没有错误。

4.3 概念结构

意义是如何构建的？内部与外部激励如何平衡？

4.3.1 维度：概念完整性与开放性

编程系统的演变使其逐渐远离概念完整性，转而形成一个由专业技术和行业标准构成的复杂生态系统。任何试图将这一生态系统的部分内容整合为一个连贯整体的做法，都会与其余部分不兼容，从而成为采纳的主要障碍。为了争取采纳，设计者不得不专注于局部的渐进式改进，并将其限制在现有实践所确立的边界之内。这使他们陷入两难：究竟能在多大程度上重视概念上的优雅，又能在多大程度上向社会施加的压力开放。我们将更详细地探讨这两个相反的方向——完整性与开放性。

4.3.2 示例：概念完整性

> 我将主张，概念完整性是系统设计中最重要的考量。与其拥有一个包含许多优秀却彼此独立、缺乏协调的构想的系统，不如让系统舍弃某些反常的功能和改进，但体现一套统一的设计理念。（Fred Brooks，《贵族制、民主制与系统设计》 [8]）

> 概念完整性并非（仅仅）源自一个人，或少数彼此认同、心意相通的人，而是有时也源自那些不为人所知的共同作者，以及设计物本身。（Richard Gabriel，《作为设计者的设计》 [27]）

概念完整性致力于从源头降低复杂性；它采用统一的概念，这些概念可以正交组合，从而创造多样性。这种方法的集大成者，或许可以在早期的 Smalltalk 和 Lisp 机器中找到：它们是围绕单一语言构建的完整编程系统。它们还纳入了通常由操作系统和数据库在编程语言之外提供的功能。所有事情都用一种语言完成，因此所有内容都以该语言的数据类型表示。同样，该语言的库和惯用法也适用于所有场景。拥有一种通用语言，避免了多语言系统固有的许多摩擦和阻抗失配。Python 编程语言也秉持着类似的理念：它遵循“应该有一种——最好只有一种——显而易见的方式来解决问题”这一原则，以推动社区就单一、一致的风格达成共识。

除了 Smalltalk 和 Lisp，许多编程语言都专注于一种数据结构 [81]:

- 在 COBOL 中，数据由嵌套记录组成，如同商业表单中的内容。在 Fortran 中，
- 数据由并行数组组成。在 SQL 中，数据是带有键约束的一组关系。在 Python、
- Ruby 和 Lua 等脚本语言中，大量数据采用字符串索引哈希表的形式。
-

最后，许多语言是命令式的，贴近可寻址内存的硬件模型，只对其稍作抽象，将其表示为原始值，以及指向可变数组和结构体的引用。另一方面，函数式语言隐藏引用，并将一切视为不可变结构化值。这种概念上的简化有利于某些类型的编程，但当命令式方法更为自然时（例如处理外部输入/输出），反而可能适得其反。

4.3.3 示例：概念开放性

*Perl，与 Python 相对照。*与 Python 的理念相反，Perl 宣称“解决问题不止一种方法”，并自认为是“第一个后现代编程语言” [91]。“Perl 完全没有任何议程，唯一的目标就是尽可能满足尽可能多人的需求。它要成为互联网以及其他一切事物的胶带。”Perl 的做法是接受演化而来的混乱现状，并借助胶带和巧思在此基础上构建。走向极端时，编程系统不再严格意义上是一个系统，而更像是一套用于即兴拼装现成软件的工具包。Perl 可谓倡导多元主义、兼容性或概念开放性的价值观，而非概念完整性。这种哲学被称为后现代编程 [66]。

*C语言++，与 Smalltalk 相反。*另一个例子是 C++：它在 C语言中加入了 Smalltalk 开发的面向对象概念，同时仍与 C语言保持 100% 兼容，甚至在应用二进制接口和性能层面也是如此。这一策略极大地促进了采纳，但代价是其复杂性远高于从零开始为面向对象而设计的语言，例如 Smalltalk、Ruby 和 Java。

更糟，反向更好。Richard Gabriel 在其颇具影响力的 1991年 文章 *Worse is Better* [26] 中首次描述了这一两难困境，分析了 Lisp 被 UNIX 和 C语言击败的原因。因为

UNIX 和 C语言极易移植到新硬件上，因此尽管它们“只能提供操作系统和编程语言所需功能的约 50%–80%”，却仍是“终极计算机病毒”。它们的概念开放性意味着能够轻松适应外部世界不断变化的条件。代价则是概念完整性下降，例如 C语言的未定义行为、杂乱无章的工作目录，以及为提供完整的开发环境而大量涌现的专用编程语言。

*UNIX*与文件。许多编程语言和系统会以“细粒度”施加结构：细化到单个变量以及其他数据和代码结构。相反，UNIX 和万维网这类系统对程序表示事物的方式限制较少。UNIX 只规定了“大型对象”这一基本基础设施[45]，，将所有细粒度结构都交由客户端程序处理。这在概念开放性方面颇具优势。文件为读写字节流提供了通用的应用程序接口；字节流是一种底层构造，具有诸多自由度，因此能够支持各种格式和生态系统。进程同样为机器级内存和处理器提供了一层薄薄的抽象。

这种开放性必然要以牺牲概念完整性为代价；尽管“万物皆文件”的说法体现了完整性，类似于 Smalltalk 的“万物皆对象”，但例外却层出不穷。目录是具有特殊操作的特殊类型文件，硬件设备文件需要特殊的ioctl 操作，而许多命令都要求文件包含换行符分隔符。此外，由于客户端程序必须为细粒度数据和代码提供自身的结构，因此它们几乎无法实现相互兼容性。结果，这些程序往往会演变成彼此竞争、各自重复建设基础设施的孤岛 [45, 44]。

万维网。与此同时，万维网 HTTP 端点已证明，这是一种比 UNIX 文件更加灵活、传播力更强的抽象。它们与文件处于相似的抽象层级，但支持更丰富的内容，并涵盖自主系统之间遍布互联网的交互。从某种意义上说，HTTP GET 和 PUT 已成为互联网规模的编程系统中的“子程序调用”。万维网最显著的特点或许在于，它的实用性让所有参与设计或与之竞争的人都大感意外。万维网很可能因为沿用了现有的文件传输实践，而在竞争中胜过了 Xanadu [65] 这样更具雄心、也不那么为人熟悉的超文本项目。

在兼容性与完整性之间做出的选择，与实用主义和理想主义这两种人格特质相关。接受技术现状并尽力善用它，是一种实用主义的做法。相反，理想主义者愿意挑战惯例、冒着遭到拒绝的风险，以实现更高远的目标。我们不禁要问，究竟是设计决策先出现，还是人格特质先形成？Lisp 和 Haskell 是教会人们进行更抽象、更连贯的思考，还是筛选出了原本就具备这种特质的人？同样，内向型开发者或许更喜欢 Smalltalk 或 Lisp 的幽静天地，而非万维网那充满冒险的“狂野西部”。

4.3.4 维度：可组合性

简而言之，通过组合若干较小的步骤，你可以到达任何地方。存在一些构建模块，可以涵盖一系列实用的组合方式。可组合性在某种意义上是“可编程性”这一概念的关键，

而且每个可编程系统都具有一定程度的可组合性（例如，在脚本语言中）。

UNIX 命令行 shell 命令是可组合性的典型例子。原始命令的基础集合可以通过用其他语言编写命令可执行文件来扩充。给定一些基本元素，可以将一个命令的输出“管道传输”到另一个命令的输入（|），按顺序执行（；或&&），通过条件进行选择，并借助循环结构重复操作，从而实现完整的命令式编程。此外，命令组合可以打包成一个命名的“脚本”，其接口与原始命令相同；脚本中也可以定义命名子程序。

在*HyperCard*中，创作环境无法组合用于编程按钮：可供选择的只有一组预定义行为。只有在编程环境中才能编写完整脚本。

Haskell 类型系统与其他函数式编程语言的类型系统一样，具有很强的可组合性。可以通过多种方式基于现有类型定义新类型，包括记录、可区分联合体、函数类型和递归结构（例如，将列表定义为空值，或由一个元素与另一个列表组合而成）。C 编程语言也提供了一些类似的类型组合方式，例如结构体、联合体、枚举，甚至函数指针。每种类型都有相应的“指针”类型。不过，它缺少 Haskell 类型所支持的递归结构。

4.3.5 维度：便利性

简而言之，只需一步，你就能得到 *X*、*Y* 或 *Z*。针对具体问题，已有现成的解决方案，但它们未必能够泛化或组合。便利性往往体现为“规范化”的解决方案和工具，以庞大的标准库形式提供。

没有便利性的可组合性，就像一组原子或齿轮；理论上，人们想要的任何东西都可以用它们构建出来，但必须亲自动手。这种情况被批评为*Lisp* 诅咒 [92]。

兼具便利性的可组合性，则是一组方便实用的专用工具以及足以构建新工具的组件。这些专用工具本身可以透明地由这些构件组合而成，但这并非必要。它们让用户省去为常见任务“自行打造”解决方案所需的时间和精力。

例如，我们来看看*UNIX* 命令行 shell 的便利性因素；前面已经讨论过它们的可组合性。原则上，所有信息都可以通过标准输入传递给程序。然而在实际使用中，为了方便，通常会采用命令行参数这一标准接口，与程序通过标准输入接收的内容分开。大多数编程系统同样兼具可组合性和便利性，提供模板、标准库或其他预先打包好的解决方案；这些方案仍可作为更大型运行的一部分，以编程方式使用。

4.3.6 维度：共性

人类可以看出，数组、字符串、字典和集合都有“大小”，但软件需要被告知它们是“相同的”。这类共性可以提取出来

形成一种显式结构（“Collection”类），类似于数据库规范化。这样，无需考虑实体的具体细节，就能查询其大小：如果c被声明为 Collection，就可以直接访问c.size。

或者，也可以将其留作隐式约定。这样做前期工作较少，但会允许各个实例出现分歧，类似于数据库中的冗余。例如，Arrays 和 Strings 最终可能使用“length”，而 Dict 和 Set 则称其为“size”。这意味着，要查询一个实体的大小，就必须根据其具体类型进行分类讨论，仅仅为了将这些分歧的路径重新汇聚到它们所表示的共性上：

如果（实体是 Array 或 String）大小：=实体.length 否
则如果（实体是 Dict 或 Set）大小：=实体.size

4.3.7 示例：扁平化与因式分解

数据结构通常包含若干个可以独立变化的“组成部分”。例如，一个简单的“车辆类型”和“颜色”二元组，可能包含 (Car, Van, Train) 与 (Red, Blue) 的所有组合。在这种因式分解表示中，我们可以直接以编程方式更改颜色：pair.second =Red 或 vehicle.colour =Red。

在某些情境中，例如类名，系统可能只允许将这种多维结构表示为穷举：RedCar、BlueCar、RedVan、BlueVan、RedTrain、BlueTrain 等。系统看到的是一个扁平的原子列表，尽管人类能够看出字符串中编码的子结构。在这种情况下，我们无法通过编程简单地“将颜色改为 Red”；而是需要按以下方式逐一判断：

if (type is BlueCar) type := RedCar else if
(type is BlueVan) type :=RedVan else if
(type is BlueTrain) type :=RedTrain
...

RedCar、RedVan、BlueCar 等之间的共性 已经被扁平化。这里存在一种尚未因式分解的隐式结构，类似于数字既可以表示为单个表达式（16），也可以表示为因数乘积（2、2、2、2）。对这种共性进行因式分解，便能还原出原始设计：其中有一对来自不同集合的值。

在关系数据库中，规范化与冗余之间存在对立关系。为了将多表数据纳入扁平的表结构中，数据需要被复制成冗余副本。当数据被尽可能分解到小表中，使每条数据都只有一个“存放”位置时，数据库就处于范式或规范化状态。冗余对于只读过程很有用，因为无需基于公共键将不同的表连接起来。然而，写入会变得有风险；为了修改一项内容，必须将更改同步到其存储的多个位置。这使得高度规范化数据库针对写入而非读取进行了优化。

4.3.8 备注：历史的终结?

如今，我们生活在一个软件技术高度发达的世界。据估计，Linux 的开发投入了 41,000 人年。我们从专业工具栈的角度来描述软件开发技术，每种工具栈可能

耗费超过 100 人年的开发时间。编程系统已演变为编程生态系统：它们并非经过设计，而是逐渐演化而来。面对现有技术这座庞然大厦，我们如何才能显著改善编程？人们有充分的动机专注于局部、渐进式的改进，而不跨越既定边界。

计算的历史是一系列演进与革命的循环。每一轮循环都先后由大型机、小型机、工作站、个人计算机和万维网主导。每次转型都会构建一个全新的技术生态系统，取代此前的生态系统，或建立在其之上。上一次革命——万维网——发生在25年前，结果是许多人从未经历过颠覆性的平台转型。历史是否已经停滞，还是我们只是陷入了一个漫长的周期，变革的压力正不断积聚？如果是后者，那么如今遭到排斥的互不兼容的理念，或许终将蓬勃发展。

4.3.9 参考文献

- 如何设计优秀的 API 及其重要性 [6]

4.4 可定制性

程序存在于系统中之后，如何对其进行扩展和修改？

编程是一个循序渐进的过程。我们可以从零开始，也可以从现有程序出发，逐步扩展和完善，直到它满足特定目的。使用不同编程系统创建的程序，可以在其生命周期的不同阶段，以不同方式完善到不同程度。

来看三个例子。首先，使用 Java 等传统编程语言编写的程序，只能通过修改其源代码来完善。不过，你或许只需添加新代码（例如新的接口实现）就能做到这一点。其次，电子表格可以随时通过修改其中的公式或数据来更改。它没有单独的编程阶段。不过，你必须直接在单元格中修改公式——无法通过指定一种独立于单元格之外的更改方式来修改它。第三，像 Smalltalk 这样的自我维持型编程系统，并未明确区分“编程”和“使用”阶段，而且可以通过系统自身对其进行修改和扩展。它赋予开发者在系统中进行实验的能力，原则上还可以从系统内部将其替换为更好的系统。

4.4.1 维度：定制的阶段

对于区分不同阶段的系统（例如编写源代码与运行程序），每个阶段的定制方法可能有所不同。在传统编程语言中，定制是在编程阶段通过修改或添加源代码来完成的，但创建的程序在运行后，就没有（自动提供的）方式可以对其进行定制。

关于定制分阶段，有许多值得探讨的问题。首先，用于定制的符号表示是什么？它可能是最初创建程序时所用的符号表示，但系统也可能使用次级符号来

进行定制（例如 Emacs 使用 Emacs Lisp）。对于具有阶段区分的系统，一个重要的问题是，这类更改是否持久。

Smalltalk、*Interlisp* 及类似系统。 在基于映像的编程系统中，通常没有严格的阶段区分，因此程序在运行期间也可以像在开发期间一样进行定制。程序映像包含编程环境。程序的用户可以打开它，导航到合适的对象或类（它们充当可寻址的扩展点），并对其进行修改。基于 Lisp 的系统（如*Interlisp* ）也遵循类似的模型。直接在映像中进行的更改会持久保存。用于 Lisp 的 PILOT 系统 [89] 提供了一种交互式方式，可在程序运行期间发生故障时进行修正。这些修正随后会应用到映像中，因此也会持久保存。

文档对象模型（*DOM*）和 *Webstrates*：在万维网编程中，传统上存在阶段区分，即编程（编写代码和标记语言）与运行（显示页面）彼此分离。不过，也可以通过浏览器开发者工具修改 DOM——可以手动修改、在控制台中运行脚本，或使用 Greasemonkey 这样的用户脚本管理器。这些更改在浏览器的默认状态下不会持久保存，但 Webstrates [50] 会同步服务器与客户端之间的 DOM，从而使更改得以持久保存。这使 DOM 支持协作，但由于事件处理涉及复杂问题，它并不会（自动）实时更新。

4.4.2 维度：寻址与可外部化性

所有编程系统中的程序都有一种表示，可以通过源代码等符号表示将其公开。定制程序时，一个值得探讨的问题是：定制是否必须通过修改原始表示来完成，还是可以通过在原有结构旁边添加内容来实现。

为了支持通过添加来进行定制，编程系统需要具备 Basman 等人 [4, 5]提出的一系列特征。首先，系统需要支持寻址：从外部引用程序表示中某个部分的能力。其次，可外部化性意味着一段已寻址状态可以在系统与外部世界之间完整传递。最后，增量式创作要求只需添加包含地址的新表达式，就能改变系统行为；换言之，任何内容都可以被覆盖而不必删除。地址如何指定，以及它们能够指向程序中的哪些扩展点，尤其重要。系统可以提供自动机制，使程序的某些部分可被寻址；也可以将这项任务交由程序员完成。

层叠样式表（*CSS*）： CSS 是万维网编程系统中增量式创作的典型范例。它提供了丰富的寻址机制，其中一部分是自动的（引用标签名时），另一部分是手动的（使用元素 ID 和类名时）。对于给定的网页，只需向 CSS 文件中添加更多规则，就几乎可以修改其外观的任何方面。Infusion 项目 [3]提供了类似的可定制性机制，但针对的是行为，而不仅仅是样式。近期的编程系统 Varv [7], 也将增量式创作作为核心原则。

面向对象编程（*OOP*）与面向方面编程（*AOP*）：在传统编程语言中，定制是通过修改代码本身来实现的。OOP 和 AOP 可以通过独立于现有程序代码添加代码来实现这一点。在 OOP 中，这需要手动定义扩展点，即接口和抽象方法。随后可以通过定义一个新类来向系统添加功能（不过，若要在不修改现有代码的情况下将新类注入其中，则需要某种配置，例如依赖注入容器）。AspectJ [48] 等 AOP 系统提供了更丰富的寻址机制。特别是，它可以通过方法调用切点等方式，为特定方法的调用添加功能。这与 Pilot [89]中的增强功能类似。

4.4.3 维度：自我可持续性

对于大多数编程语言、编程系统和普通软件应用程序而言，如果希望进行超出某一程度的定制，就必须超越系统本身所提供的功能。大多数编程系统都明确区分了用户层和实现层：前者用于使用系统，后者则存放系统本身的源代码。如果用户层没有提供对某项属性或功能的控制，就只能转到实现层。在通常情况下，这会涉及一种完全不同的语言或系统，并伴随相应的学习成本。此外，它很可能处于更低的层次——缺少用户层所具备的表现力函数、功能或抽象——从而导致更加繁琐的编程体验。

然而，通过精心设计系统，可以在用户层暴露其实现的更深层面，从而放松此前对这些层次的严格划分。例如，在研究系统*3-Lisp* [83], 中，通常内置的条件if 函数和错误处理catch 函数，都是以用户层的 3-Lisp 代码实现的。

系统内部运作对用户层的可访问程度，我们称之为自我可持续性。在这一维度的极致一端，是类似“干细胞”的系统：无需“跳出”系统、转而进入更低的实现层，这类系统便能逐步演化出任意行为。从某种意义上说，这些系统之间的差异仅仅是初始状态不同，因为任何一个系统都可以转变成其他任意一个系统。

另一端，即自我可持续性最低的一端，对应着最低程度的可定制性：除了构成任何软件用户层的瞬态运行时状态变化外，用户若不转而使用系统的实现手段，就无法改变任何东西。这类似于传统的面向狭窄领域的终端用户“应用程序”，无法用于其他用途。

“自描述”或“自实现”这两个术语曾被用来描述这一特性，但容易引起困惑：一件事物怎么能描述自己？相比之下，能够自我维持 的系统更容易理解。我们所见到的自我可持续性很强的例子，往往都类似操作系统。UNIX 被广泛视为操作系统，而 Smalltalk 和 Lisp 则被冠以不同的称谓。尽管如此，这三者都曾作为定制硬件的操作系统推出，并且

承担着相似的职责。具体来说：它们支持“程序”的执行；定义用于访问和修改状态的接口；提供常用功能的标准库；规定程序之间的通信方式；并提供用户界面。

UNIX：UNIX 的自我可持续性得益于两个因素的结合。首先，系统以二进制文件（通过可执行与可链接格式[13]）和文本文件（用于配置）的形式实现。其次，这些文件属于面向用户的文件系统，因此用户可以通过 UNIX 文件接口替换和修改系统的部分内容。

Smalltalk 和组合对象 *Lambda* 架构：Smalltalk 的自我可持续性与 UNIX 类似，但粒度更细，也较少强调内容存储在易失性（进程）存储还是非易失性（文件）存储中。类似之处在于：（1）系统由对象构成，对象的方法中包含 Smalltalk 代码；（2）这些内容可以通过类浏览器和代码编辑器修改。组合对象 Lambda 架构（Combined Object Lambda Architectures，简称 COLA） [75], 是一种理论上的系统设计，旨在提升 Smalltalk 的自我可持续性。其实现方式是推广对象模型，使其支持类之外的关系。

4.4.4 参考文献

除上述讨论的例子外，自维持系统研讨会的论文集 [41, 40] 还提供了许多系统和语言的例子，它们能够引导自身启动、实现自身、修改自身并维护自身；Gabriel 对编程语言革命的分析 [28] 以 PILOT 中的*advising* 、相关的 Lisp 机制以及 OOP 中的“混入”为例，说明“语言”和“系统”两种范式之间的区别。

4.4.5 关系

- 展平与分解（第4.3.7节）：二者相关，因为“可定制性”是一种从现有程序创建新程序的方式；将重复性内容分解到可复用标准组件库中，也有助于实现同样的目标。交互（第4.1节）：这决定了运行程序和编写程序是否分属不同阶段，因此
- 可能影响可进行何种定制。

4.5 复杂性

系统如何组织复杂性？需要达到何种细节层次？ ?

计算机执行的是一系列低级指令，而人类对程序的描述则使用更高层的术语；两者所需的细节层次之间存在巨大差距。为了弥合这一差距，编程系统需要处理从高层描述转化为低级指令过程中固有的复杂性。

自 20 世纪 40 年代以来，程序员们一直设想，“自动编程”将实现更高层次的编程。这并不一定意味着完全自动化。

[13]可执行与可链接格式。

事实上，最早的“自动编程”系统指的是配备编译器（或解释器）的高层编程语言，可将高层代码扩展为详细指令。

大多数编程系统都会分解复杂性，将需要指定的一些细节封装成程序员可以重复使用的组件。这些细节可能封装在程序库中，也可能由编译器或解释器补充。这样的分解也可能反映在系统的概念结构中（第 4.3.7 节）。不过，系统也可能自动完成 编程的某些方面。在这些情况下，一个通用算法可以解决整类问题，因此无需显式编写代码。可以想想 SQL 查询的执行规划，或为 Prolog 这类逻辑编程语言提供支持的推理引擎。

4.5.1 备注：表示法

即使是在高层进行编程，也需要处理某种程序符号表示。在高层函数式或命令式编程语言中，程序员编写的代码通常具有明确的操作语义，即使部分复杂性被交由程序库实现或运行时处理也是如此。使用 SQL、Prolog 或 Datalog 这类声明式编程系统时，程序的含义仍然明确，但其定义并非基于操作过程——系统会通过一个（或多或少具有确定性的）推理引擎，根据给定的描述来解决问题。最后，基于示例编程 的系统则离明确的操作语义更远了一步——程序可能只是一组示例输入和输出，系统引擎据此推断出具体的执行步骤（推断过程可能具有非确定性）。

4.5.2 维度：复杂性分解

应对复杂性的基本机制是对其进行因式分解。给定一个程序，逻辑中越是领域特定的方面，就越需要明确指定；而越是常规且技术性的方面，则越交由可复用组件处理。通常，这个可复用组件只是一个程序库。不过，对于更高级的编程语言，可复用组件可能包含语言运行时的一部分，例如内存分配器或垃圾回收器。对于声明式语言或示例编程，可复用组件则是一个通用推理引擎。

4.5.3 维度：自动化程度

复杂性分解可以让程序员免于处理某些细节，但这些细节仍然需要显式编程。根据系统的可定制性，这种编程可能可以访问，也可能无法访问，但它始终存在。例如，电子表格公式中使用的函数是在电子表格系统中实现的。

自动化程度较高的编程系统，需要的不只是将代码分解为可复用的组件。它采用一种机制，不必明确指定程序操作语义中的某些细节，而由系统自动推断。这就是示例编程和

机器学习的方法，通过示例来指定行为。在某些情况下，要判断某项功能属于自动化还是仅仅属于复杂性分解，并不那么明确：垃圾回收既可以被视为一种简单的自动化，也可以被视为一种复杂的复杂性分解。

这里还存在一种有趣的（也许是不可避免的）权衡。自动化程度越高，程序的操作语义就越不明确。这会带来广泛的影响。例如，Smaragdakis [82] 指出，这意味着实现方式可能会显著改变程序的性能。

4.5.4 示例：领域特定语言

领域特定语言 [22] 提供了一个不涉及自动化的复杂性因式分解示例。在这种情况下，编程分为两个层次。在较低层次，一名（通常经验更丰富的）程序员开发一种领域特定语言，使另一名（通常经验较少的）程序员能够轻松解决特定领域中的问题，例如建模金融合约或指定交互式用户界面。

领域特定语言提供了可以组合的基本元素，但每个基本元素以及每种组合形式都具有经过明确编程且无歧义的操作语义。领域特定语言的用户可以用它所提供的更高层次概念来思考，而这种概念结构可以使用第 ~43 节中的维度进行分析。只要这些概念是清晰的，用户就无需关注生成的程序究竟如何运行的细节。

4.5.5 示例：示例编程

自动化的一个有趣案例是示例编程 [58]。在这种情况下，用户甚至不提供关于程序行为的声明式规范，而是指定示例输入和输出。随后，某种或多或少复杂的算法会尝试推断输入与输出之间的关系。例如，这可以通过程序合成来实现：算法使用少量预定义操作组合出一种变换。示例编程通常非常易于使用，并已应用于电子表格应用 [35]中。

4.5.6 示例：更高阶的自动化

纵观历史，程序员始终期待着“自动编程”的更高层次。正如Parnas [70], 所指出的：“自动编程一直是一个委婉说法，指的是使用比当时程序员可用的语言更高层次的语言进行编程”。

我们可以推测，深度学习是否会促成自动化的下一步发展。不过，从原理上说，这与现有的发展并无不同。我们可以认为，任何自动化程度都运用了人工智能 方法。声明式语言或基于约束的语言便是如此——其推理引擎实现了一种传统的人工智能方法（GOFAI，即传统人工智能）。

4.5.7 关系

- 概念结构 （第 4.3 节）：在许多情况下，复杂性分解遵循编程系统的概念结构。
  展平与分解 （第 4.3.7 节：通常，人们会自动化因式分解中处于最低层级的事物
- （方法是让最低层级成为程序之外的事物——位于系统或程序库中）

4.6 错误

系统认为什么是错误？如何预防和处理这些错误?

计算机系统并不了解人类的意图。人类总会犯下系统无法识别为错误的差错。尽管如此，系统能够识别的差错仍有很多，而系统的设计将决定哪些人类差错能够成为可检测的程序错误。这涉及几个问题：什么会导致错误？哪些错误可以预防？系统应如何应对错误?

遵循关于错误的标准文献 [77], ，我们区分四种错误：失误、疏忽、差错和故障。失误是由短暂的人类注意力失效导致的错误，例如源代码中的打字错误。疏忽与之类似，但由记忆失效导致，例如错误地记住了方法名。差错是逻辑错误，例如算法设计不当。最后，故障是由系统自身导致、程序员无法控制的系统错误，例如硬件或虚拟机故障。

4.6.1 维度：错误检测

系统实现的任何反馈循环 中都可以识别错误。根据反馈循环的性质，这既可以由人来完成，也可以由系统本身完成。

来看三个例子。首先，在实时编程系统中，程序员可以立即看到代码更改的结果。错误检测由人来完成，系统则可以通过尽可能多地可视化代码更改带来的后果来提供帮助。其次，在具有静态检查反馈循环的系统中（例如语法检查、静态类型系统），分析结果会报告潜在错误。第三，开发的软件运行时也可以检测错误：无论是程序员测试软件时（手动测试或通过自动化测试），还是用户运行软件时。

不同反馈循环中的错误检测适用于检测不同类型的错误。静态检查反馈循环可以检测许多失误和疏忽，但并非总是如此。例如，考虑一种“紧凑”的表达式分布，在这种情况下，代码的微小变化可能会导致行为发生巨大变化。这使得失误和疏忽更容易造成难以检测的错误。通过实时反馈循环更容易检测差错，但更先进的静态检查也能在一定程度上检测出这些差错。

4.6.2 示例：静态类型

在 Haskell 和 Java 等静态类型编程语言中，类型用于捕捉有关程序员意图的某些信息。类型检查器会确保代码符合通过类型给出的轻量级规范。在这类系统中，类型和实现是对程序员意图的两种描述，二者需要保持一致；不同之处在于类型能够捕捉意图的程度，以及二者的构建方式，也就是说，二者中哪一个先出现。

4.6.3 示例：测试驱动开发、交互式解释器与实时编程

静态类型旨在不执行代码的情况下检测错误，而基于即时反馈的方法通常会执行（部分）代码，让程序员立即看到错误。这可以通过多种方式实现。

对于测试驱动开发，测试充当规范（很像类型），据此检查实现。此类系统提供反馈的即时程度可能有所不同，具体取决于测试何时执行（在后台自动执行，或手动执行）。配备读取-求值-打印循环（REPL）的系统允许程序员即时运行代码并检查结果。为了有效检测错误，结果必须易于观察：打印输出比系统状态的隐蔽变化更有帮助。最后，在实时编程系统中，代码会立即执行，程序员识别错误的能力取决于系统状态的可观察程度。例如，在实时编程音乐中，你会听到代码并非你所期望的，从而获得一种易于使用的即时错误检测机制。

4.6.4 备注：消除潜伏错误

错误检测的一个常见目标是防止潜伏错误，也就是在执行过程中的某个较早时刻发生、但直到后来出现意外行为时才显现出来的错误。例如，我们可能会解引用错误的内存地址，并将一个无用值存入数据库；只有在访问数据库时，我们才会发现问题。不同的反馈循环可以采用不同方式来防止潜伏错误。在实时反馈循环中，可以通过可视化通常会被隐藏的效果来实现这一点。运行软件时，可以通过尽早检测错误的机制来防止潜伏错误（例如，将指针初始化为null，并在对其解引用时停止执行。）

Elm 与时间旅行调试。识别潜伏错误的一种值得注意的机制，是由 Elm 编程语言推广的时间旅行调试概念。通过时间旅行调试，程序员可以沿时间回溯，查看在某个特定时刻之前执行了哪些步骤。这样一来，当潜伏错误显现时，可以中断执行，然后沿执行过程回溯到错误的实际来源。

4.6.5 维度：错误响应

检测到错误时，系统通常有多种应对方式。以下内容适用于在执行过程中提供某种错误检测机制的系统。

- 系统可能会尽可能自动从错误中恢复。这对于较简单的错误（失误和疏忽）或许可行，但也适用于某些差错（算法并发逻辑中的差错通常可以通过重新启动代码来解决）。系统也可能继续运行，仿佛错误从未发生。这可以省去代价高昂的检查，但之后可能会导致潜伏错误。系统还可以询问人类如何解决问题。既可以通过交互方式进入可修改代码的模式，也可以通过停止系统以非交互方式处理。
-

除了上述选项之外，系统还可以通过回溯执行过程来查找根本原因，从而从潜伏错误中恢复。系统也可以提供撤销机制，撤销期间发生的所有操作，例如通过事务处理实现。

Interlisp 和按我所想执行 (DWIM)。Interlisp 的 DWIM 功能会尝试自动纠正失误和疏忽，尤其是拼写错误和括号不匹配的情况。当 Interlisp 遇到错误（例如引用未定义符号）时，就会调用 DWIM。此时，DWIM 会搜索当前用户经常使用的名称相似的符号。若找到，就会自动调用该符号、纠正源代码并通知用户。在更复杂的情况下，如果 DWIM 无法自动纠正错误，就会与用户交互，让用户手动纠正。

4.6.6 关系

- 反馈循环：错误检测总是作为单个反馈循环的一部分进行。因此，反馈循环决定了错误检测能够发生的结构。自动化：半自动错误恢复系统（例如 DWIM）实现了一种自动化形式。反脆弱软件 [64] 的概念是通过自动化实现错误恢复的一个更复杂的例子。表达式分布：在一种表达式分布中，符号表示的微小变化会产生有效但行为不同的程序，因此，小失误或疏忽更有可能导致难以通过标准机制检测的错误。
-

4.6.7 参考文献

传统编程语言中最常见的错误处理机制是异常处理。Goodenough [31] 描述了现代形式的异常处理；Ryder 等人 [79] 记录了软件工程对异常处理的历史影响。反脆弱软件 [64] 这一概念更进一步，指出软件可以在错误的作用下得到改进。混沌工程 [10]的相关工作正朝着这一方向迈进。

Reason [77] 在人为错误的背景下分析错误，并提出了一种错误分类法，我们采用了这一分类法。在计算领域，错误或错误计算也曾从哲学角度进行分析 [23, 20]。值得注意的是，不同编程亚文化对错误的态度和处理方式也各不相同 [73]。

4.7 可采纳性

该系统如何促进或阻碍个人和社区的采纳?

我们将个人的采纳视为易学性维度，将社区的采纳视为社交性维度。

4.7.1 维度：易学性

主流软件开发技术需要投入大量精力才能学会。可以通过以下几种方式让系统更易于学习：

- 针对特定应用领域进行专门设计。针对简单的小规模需求进行专门设计。利用特
- 定社区的背景知识、技能和术语。通过分阶段的复杂性等级和辅助开发工具支持
- 学习 [24]。更好的反馈循环会有所帮助（第4.1节）。将异构技术栈整合为更简单、统一的系统。这与概念结构下的维度有关（第4.3节）。
-

-

FORTRAN 是编程领域的一项突破，因为它专注于科学计算，并利用科学家对数学公式的背景知识。相比之下，COBOL 专注于商业数据处理，通过摒弃数学、采用通俗英语，拥抱商业社群。

LOGO 是第一种专为教儿童而设计的语言。后来，BASIC 和 Pascal 被设计出来，用于在大学阶段教授当时标准的编程概念。到了 90 年代，BASIC 和 Pascal 又在微处理器上迎来了第二春。这些微处理器编程系统以完整的解决方案著称，整合了所需的一切，因此成为一代程序员的家庭学校。近来，Racket、Pyret 和 Grace 等语言通过分阶段展现逐步递增的复杂性，支持学习。Scratch 则回归了 LOGO 教授儿童的理念，采用强调趣味性而非通用性的图形化编程环境。

有些编程语言有意识地优先考虑程序员学习和使用它们时的体验。Ruby 将自己称为*程序员最好的朋友*，专注于简洁与优雅。Elm 面向 Web 应用这一范围较窄但仍相当广泛的领域，同时注重简洁和对程序员的友好性。它放弃了可能导致运行时崩溃的功能，也竭力让错误消息清晰且便于采取行动。

如果我们超越编程语言*本身*来看，就会发现一些易学性更好的可编程系统。最好的例子是电子表格，它提供了一种更简单、更直观的专用计算环境。网格这一视觉隐喻利用了人类的感知技能。将所有编程都转移到声明式公式和属性中，大大简化了创建和理解过程。实时编程研究 [36, 90] 一直试图将这些优势融入通用编程，但迄今为止成效有限。

HyperCard 和 Flash 都是被非专业人士广泛采纳的编程系统。与电子表格一样，它们都有一种组织信息的视觉隐喻（分别是卡片和时间线）。它们都让初学者很容易上手。HyperCard 还设置了多个复杂性层次，旨在促进逐步掌握。

Smalltalk 和 Lisp 机器虽然复杂，却是统一的。克服最初的学习曲线后，它们的环境便能为构建任意复杂度的完整应用系统提供完整解决方案，无需学习其他技术。Boxer [15] 尤为值得一提，它提供了一个通用编程环境——尽管仅适用于小规模应用——以及类似电子表格的组织性视觉隐喻。

4.7.2 维度：社交性

随着时间推移，尤其是在互联网时代，社会因素逐渐成为编程的主导议题。如今，许多编程技术由开源社区开发，而所有编程技术也都融入了其用户的社交媒体社区。因此，影响社交性的技术决策可能至关重要 [61]。其中包括：

- 兼容性：易于集成到标准技术栈中，从而支持渐进式采纳，并且在有需要时也能轻松退出。这种动态在经典文章 *Worse isBetter* [26]中有所探讨，该文论述了 UNIX 如何击败 Lisp。采用开源方法进行开发，可以获得志愿者的劳动，并培育
- 由爱好者组成的用户社区。开源开发的技术优势，最初由文章 《大教堂与集市》 [76], 广为传播；该文指出：“ 只要有足够多的眼睛，所有错误都无处藏身。” 开源已成为软件开发工具的标准，即便是大型企业内部开发的工具也不例外。通过软件包仓库或开放式交换平台，可以轻松共享代码。在开源时代之前，商业市场十分重要，例如 VisualBasic 的 VBX 组件。当语言缺少标准库时，共享就会受
- 阻，进而催生相互竞争的方言，例如 Scheme [92]。利用专门的社交媒体社区提供技术支持，有助于培育这些社区。志愿者提供的技术支持与志愿者贡献的代码一样，都能扩大核心开发者的影响力。在某些情况下，Stack Exchange 等社交媒体甚至已经取代了文档。
-

有人可能会认为，社交性并非纯粹的技术维度，因为它也涉及产品管理的某些方面。相反，我们认为，社交性是一项普遍存在、贯穿各处的议题，无法与技术分离。

围绕某个编程系统形成的在线社区，其整体氛围可能是该系统最公开可见的特征。早在社交媒体出现之前，Flash 就已拥有一个充满活力的社区，业余爱好者们在其中分享代码和技巧。Elm 语言从一开始就投入了大量精力，致力于营造一个友好的社区 [13]。改革老牌社区的尝试引入了行为准则，但也引发了不少争议。

另一方面，与外部世界隔绝的社区能够赋予其成员强烈的归属感和使命感。例如，Smalltalk、

Racket、Clojure 和 Haskell。这些社区与邪教有些相似，拥有大师型领袖以及强烈的群体凝聚力。

编程系统的经济可持续性有时甚至比严格意义上的社会和技术问题更加重要。采纳一项技术，就时间、金钱以及放弃其他机会而言，都是一项代价高昂的投资。每个人在投资于由不会消失的大型企业支持的技术，或投资于已经得到广泛采纳、因而必然能够持续存在的技术时，都会感到更加安心。一个充满活力且成熟的开源社区为某项技术提供支持，也会让人更放心。

遗憾的是，社交性常常与易学性相冲突。兼容性会使新学习者需要掌握的历史包袱不断增加。大型互联网公司主要投资于与其专家员工和高端需求相关的技术。开源社区主要围绕专家程序员为“解决自己的需求”而使用的技术蓬勃发展。尽管风险投资不断流入“无代码”和“低代码”编程系统，但目前尚不清楚它们如何才能实现经济和社会上的可持续发展。总体而言，互联网时代见证了专家程序员的崛起，以及面向“我们其他人”的编程系统逐渐黯然失色。

5 评估

应根据这些技术维度在设计和分析编程系统时的实用性来评估它们。为此，本节展示了该框架的两种用法。首先，我们使用这些维度分析近期的编程系统 Dark [56]，，说明它与以往工作的关系，以及它对技术水平的贡献。其次，我们使用技术维度找出编程系统设计空间中一个尚未探索的新位置，并构想一种可能由此分析催生的新设计。

5.1 评估 Dark 编程系统

Dark 是一种用于构建“无服务器后端”的编程系统，即供万维网和移动应用程序使用的服务。它旨在通过“消除偶然复杂性”[14] 来简化此类服务的构建；这些偶然复杂性源于其部署和运行通常涉及的大量系统，其中包括用于编排、扩缩容、日志记录、监控和版本控制的基础设施。Dark 为开发提供集成工具（图 3），并被描述为无需部署，这意味着代码可以即时部署到生产环境。

Dark 展示了我们为何需要从更广阔的视角看待编程系统。当然，它包含一种编程语言，其设计受到 OCaml 和 F# 的启发。但 Dark 的独特之处在于，它消除了部署现代云服务所需的许多辅助系统。这些系统并不属于

[14] https://roadmap.darklang.com/goals-of-dark-v2.html，访问日期：2023-02-05。

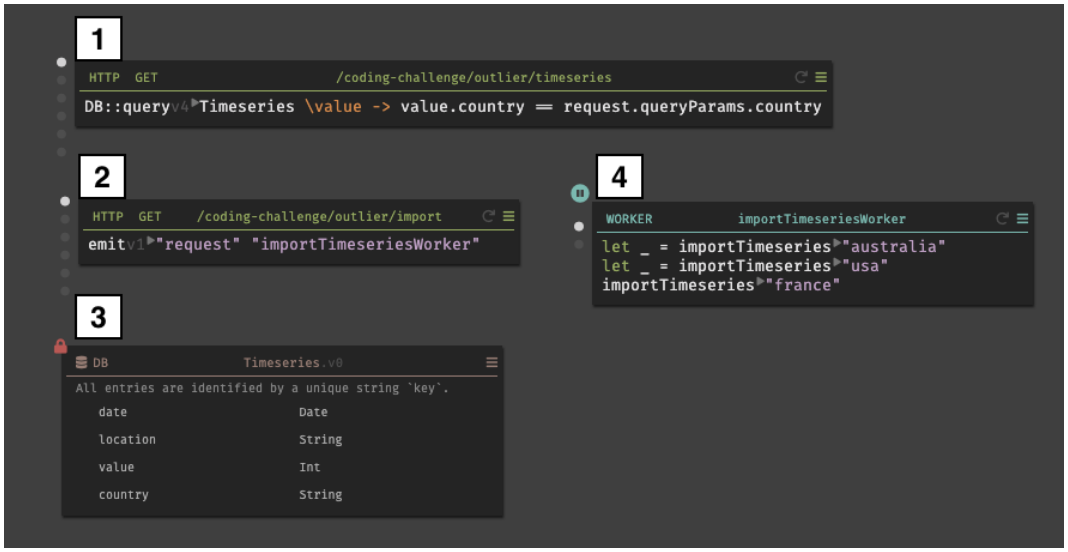


图 3 Dark 中一个简单的网络服务，由两个 HTTP 端点（1、2）、一个数据库（3）和一个工作进程（4）组成。

典型的编程语言，却构成了整个开发过程复杂性的重要部分。

借助技术维度，我们可以超越“推销辞令”，深入了解幕后情况，更好地理解 Dark 作为编程系统的有趣技术特性。表 4 和表 5 总结了下文更为详细的分析。这类分析有两个显而易见的好处：

1. 它提供了一份较具体的主题列表，可供我们在考察 Dark 这类编程系统时深入研究。2. 它为这些主题提供了一套通用词汇，让我们能够基于相同标准将 Dark 与其他系统进行比较。

5.1.1 Dark 的维度分析

交互模式与反馈循环。传统的交互模式（4.1.4）包括运行、编辑和调试。对于现代网络服务而言，运行是指在云环境中运行，而这通常还会带来其他形式的反馈（日志记录和监控）。Dark 的关键设计决策是将所有这些不同的交互模式整合为一种模式。这种紧密集成让 Dark 能够提供更即时的反馈循环（4.1.1），使代码更改能够立即生效，不仅开发者可以看到，外部用户也能看到。这种集成式交互模式让人想起 Smalltalk 中基于图像的环境；Dark 则通过在多用户云环境中采用这一模型，推动了这一领域的发展。

反馈循环与错误响应。开发与运行的整合，也使得利用运行期间出现的错误 来推动开发成为可能。具体来说，当 Dark 服务收到不受支持的请求时，用户可以构建处理程序 [11] 来提供响应——充分利用实时数据

表 4 Dark 在各项特征维度上的定位摘要。为简洁起见，维度 n-Dark 与普通编程没有区别的维度被省略。

维度（集群）摘要	
交互	
交互模式	单一集成模式包含开发、去-调试和运行（“无需部署”）
反馈循环	代码编辑由用户触发，或由不受支持的 HTTP 请求触发，变更会自动部署，从而实现即时反馈
错误	
错误响应	收到不受支持的 HTTP 请求时，程序员可以使用处理过程中的请求数据编写处理程序代码
概念结构	
概念完整性与……	特定领域高层的抽象以及功能低层的抽象都经过精心设计，以确保概念完整性。
开放性	
可组合性	用户应用程序由高层原语组合而成，性；低层使用可组合的函数式抽象，（记录、管道）。
便利性	提供强大的高层领域特定抽象（HTTP、数据库、工作进程）；低层则有核心函数式底层存在库。

作为请求的一部分发送。就我们的维度而言，这是一种**错误响应** (4.6.5)，最初由用于 Lisp [89] 的 PILOT 系统开创。Dark 不仅用这种方式响应错误，还将其作为主要的开发机制，我们可以称之为**错误驱动开发**。这样，Dark 用户便可以根据示例输入值构建程序。

概念结构与易学性。Dark 程序使用高层概念来表达，这些概念专用于服务器端 Web 编程领域：HTTP 请求处理程序、数据库、工作进程和定时任务。它们旨在减少偶然复杂性，并致力于实现高度的概念完整性 (4.3.1)。在代码层面，Dark 使用一种通用函数式语言，并着重强调某些概念，尤其是记录和管道。这些高层概念有助于提升系统的易学性 (4.7.1)，因为它们高度契合特定领域，预期用户也已经熟悉这些概念。

表 5 摘要：Dark 在其区别性维度上的定位。为简洁起见，n-
省略了 Dark 与普通编程没有区别的维度。

维度（集群）	摘要
可采纳性	
易学性	高层概念会让 目标受众感到熟悉；低层语言则具有常见的 基础函数式编程学习曲线
符号表示	
符号结构	高层概念的图形符号表示较为复杂—— 通过结构编辑器处理低级代码
一致性	数据库使用通用的符号结构 以及代码，从而能够对 顺序结构（记录、管道、表）使用相同的编辑构造
复杂性	
复杂性分解	云基础设施（部署、编排等） 由 Dark 平台提供，程序员无法看到 但也无法修改
自动化程度	当前实现提供基本基础设施， 但平台未来可以提供更高程度的自动化， 例如提高可扩展性
可定制性	
定制分阶段	系统可以在运行时进行修改，且所做的更改 会被持久化，但必须在 Dark 中进行 编辑器，与运行中的服务不同

表示结构与一致性。 Dark 结合使用图形编辑器和代码。这两种符号表示遵循互补记法 (4.2.1)模式。窗口界面用于处理高层概念，而代码用于处理低层概念。在高层，代码被组织在二维表面上可自由放置的方框中。与 Boxer 不同 [16], ，这些方框不会嵌套，且该空间不能用于放置其他内容（例如注释、架构示意图或其他媒体）。在低层，代码通过语法感知型结构编辑器进行操作，该编辑器会显示纯函数的推断类型和计算出的实时值。它还为记录和管道提供专门的编辑支持，允许用户分别添加字段和步骤。

复杂性分解与自动化。 Dark 的宣传目标之一是消除偶然复杂性。为此，Dark 将开发、云部署、编排和运行通常所需的异构技术栈整合起来。Dark 通过复杂性分解 (4.5.2)将其隐藏起来。

高级基础设施由 Dark 平台提供，并对用户隐藏。基础设施通过显式编程实现，无需复杂的自动化(4.5.3)。这种对以往需要手动编写代码的功能进行分解的做法，与高级编程语言中垃圾回收的发展遵循着类似的模式。

可定制性。 Dark 平台清楚地区分了平台本身与用户应用程序，因此，自我可持续性（ 4.4.3 ）并非其目标。平台与用户之间的严格划分（与前文所述的复杂性分解有关）意味着，要更改 Dark，就必须修改平台源代码本身；该源代码采用的许可证仅允许将其用于贡献目的。同样，应用程序本身也是通过修改和添加代码来开发的，这需要对代码进行破坏性访问，因此增量式创作 （4.4.2）在这两个层面都未体现。得益于执行与开发的整合，运行期间可以进行持久性更改（参见定制分阶段，4.4.1），但这是通过 Dark 编辑器完成的，而编辑器与运行中的服务是分离的。

5.1.2 Dark 的技术创新

这项分析揭示了 Dark 编程系统的一些有趣之处。首先，不同交互模式 之间紧密整合，消除了异构的技术栈，使 Dark 易于学习，并能从已部署服务中快速获得反馈。其次，系统利用错误响应 来指导 HTTP 处理程序的开发。借助技术维度框架，可以更准确地描述这两点，也可以探讨如何在其他编程系统中实现这些特性。该框架还指出了可能的替代方案（或许也能带来改进），例如构建一个具有 Dark 类似特征、但允许从系统内部更灵活地修改平台且更具自我可持续性的系统。

5.2 探索设计空间

稍加处理，便可将技术维度的取值绘制成简单的散点图，从而让我们看出设计空间中的模式或空白。在此，我们将考察以下编程系统的两个维度：符号表示多样性[15] 和自我可持续性：Haskell、Jupyter 笔记本、Boxer、HyperCard、万维网、电子表格、Lisp、Smalltalk、UNIX 和 COLA。

虽然我们将维度描述为定性概念的做法是提出这些维度的必要条件，但若要像这样将它们的关系可视化，显然也需要某种生成数值的方法。为求简便，[16] 我们采用以下方案。对于每个维度，我们将其核心思想提炼为若干个是非问题（附录A）

[15]这就是我们所称的符号表示的一致性（4.2.10）这一维度，只不过方向相反。[16]毫无疑问，可以通过许多方法将我们的描述转化为各种度量；这些方法各有优缺点，适用于不同的目的，但这超出了本文的讨论范围

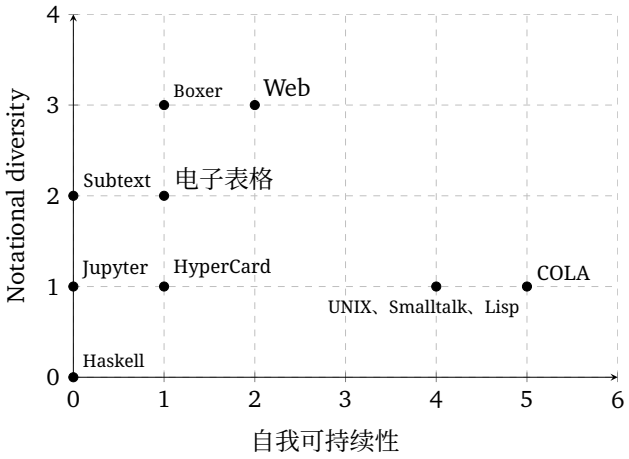


图4 示例编程系统（或系统家族）在自我可持续性和符号多样性方面的测量结果，揭示了同时具备两者较高程度的系统并不存在。

这些问题能够充分体现我们观察到的、希望绘制出来的系统之间的差异。然后，对于每个系统，我们将回答“是”的数量相加，得到该维度一个合理的评分。

图4展示了我们通过问题集合获得的结果。它表明，以应用为重点的系统涵盖了一定范围的符号多样性，但其自我可持续性仅处于相当低的水平。“类操作系统”（第3.2节）聚集在右侧的一个“岛屿”中，其符号多样性完全相同，自我可持续性也几乎相同。右上方还有一片醒目的空白区域，代表一种尚未探索的组合，即两个维度都具有较高值。对于其他维度组合，我们或许可以将其视为存在对立关系的证据，即其中一个维度更高，往往意味着另一个维度必然更低（也许可以寻找一个能够更好描述这种关系的单一新维度）。不过在本例中，拥有大量表示法与能够从内部改变系统之间并不存在明显冲突。因此，我们将这一空白解读为一个值得尝试的新机会：将COLA的自我可持续性与Boxer及Web开发的符号多样性结合起来。事实上，这或多或少就是主要作者即将完成的博士论文。

6 结论

人们重新燃起了开发新型编程系统的兴趣。这类系统超越了这样一种简单模式：使用或多或少功能强大的文本编辑器，在编程语言中编写代码。它们结合文本和视觉符号表示，通过丰富的图形交互创建程序，并对有关程序编辑、执行和调试的既有假设提出挑战

论文。在这里，我们只是想证明这是可能的，并展示人们可以如何利用这些结果。

。尽管新型编程系统的数量不断增加，但要评估编程系统的设计，并了解它们相较于以往工作有何改进，仍然很困难。为解决这一问题，我们提出了一个“技术维度”框架，以定性而严谨的方式概括编程系统的基本特征。

技术维度框架让过去和现在种类繁多的编程系统能够置于同一基础上进行比较。这对于下一代编程系统的设计者识别各个系统的优势，并在可能的情况下将其结合起来，至关重要。随着越来越多的系统依据该框架接受评估，可能性空间的图景将逐渐显现。一些区域会明显空缺，表明尚未实现的可能性，而这些可能性或许值得尝试。这样一来，便为编程系统的设计创建了一个“常规科学”领域。

致谢 我们特别感谢 Richard Gabriel 为我们提交给 2021 年编程语言模式 （PLoP）会议的论文担任指导人。我们感谢 PLoP 写作研讨会的参与者提供的反馈，也感谢其他在不同阶段校对论文或就其中观点提出意见的人士。他们包括 Luke Church、Filipe Correia、Thomas Green、Brian Hempel、Clemens Klokmose、Geoffery Litt、Mariana Mărășoiu、Stefan Marr、Michael Weiss，以及 Rebecca 和 Allen Wirfs-Brock。我们还感谢参加 Programming 2021 对话引导环节的与会者，以及参加我们在 Programming 2022 举办的教程/研讨会的与会者；该教程/研讨会的名称是“编程系统方法论”（MOPS）。

本研究部分得到了捷克科学基金会项目（编号 23-06506S）的支持。

A 使维度量化

为了生成< стиль id='1'>自我可持续性 和符号多样性的数值坐标，我们将这两个维度分别拆分为少量“是/否”问题，并统计每个系统的“是”答案数量。我们以非正式的方式提出这些问题，目标是实现以下三点：

1. 捕捉该维度的基本理念或特征 2. 使先前的印象更加精确（即大致符合我们凭直觉判断某些关键系统所处的位置，同时为我们不太确定的系统提供精确度并带来可能的意外。） 3. 在达到上述目标所必需的前提下，问题数量尽可能少

这一过程的结果如下，并简要说明了每个问题的理由。之后，我们会以一些备注作结。

A.1 自我可持续性

问题：
1. 无需重启，就能向系统命名空间中添加新项目吗？JavaScript 是典型的例子，其中 Array 或 Object 这类“内置”类可以随意扩展（也可以被破坏性地修改，不过这是另一个问题）。具体来说，如果用户希望让所有数组都能使用新的 sum 运行，就可以直接将该方法添加到 Array 原型中，就像它只是一个普通对象（它确实如此），而不会因此受到阻止。如果必须重新编译甚至重启系统，就无法在系统内部有意义地实现这一点。反过来说，能够做到这一点意味着普通程序也可以修改“内置”命名空间，这表明实现层与用户层之间的界限不那么明显，而且这似乎对自我可持续性很重要。

2. 程序能否生成程序并执行它们？这一特性与“代码即数据”或存在 eval() 函数有关，是实现自我可持续性的关键要求。否则，除了从预定义的行为列表中进行选择之外，要重新编程该系统，就必须编辑外部表示并重新启动系统。如果用户可以在系统内部输入文本，那么他们就能够编写代码——然而，除非系统能够将内部数据结构解释为程序并实际执行它们，否则这些代码将不会产生作用。

3. 更改是否具有足够的持久性，从而鼓励无限期的演化？如果最初的尝试或后续进展可能因意外关闭窗口而被重置，或者只能通过复杂的流程保存，那么这会阻碍从系统内部对其进行任何长期改进。例如，在使用网页浏览器开发者工具开发 JavaScript 应用程序时，可以在控制台中运行任意 JavaScript，但这些更改只适用于正在运行的实例。在控制台中利用具体的系统状态进行尝试之后，仍然必须返回源代码文件，并手动进行相应更改。刷新页面以加载更新后的代码时，系统会从全新的初始状态开始。这意味着，除了尝试之外，不值得使用正在运行的系统进行任何编程。

4. 您能在运行中的系统内重新编程底层基础设施吗？这或许忠实地总结了 COLA 的工作如何旨在在这一维度上超越 Lisp 和 Smalltalk。

5. 用户界面能否在系统内部任意更改？无论是否将其归类为“底层基础设施”，系统的视觉和交互方面都是系统的重要组成部分。因此，要称其真正具备自我可持续性，这些方面就需要像系统的其他部分一样，能够开放地进行重新编程。

问题												
1							✔	✔	✔	✔	✔
2							✔	✔	✔	✔	✔
3			✔		✔	✔		✔	✔	✔	✔
4											✔
5								✔	✔	✔	✔
总计	0	0	1	0	1	1	2	4	4	4	5

A.2 符号多样性

问题：
1. 文本符号表示是否有多种语法？显然，拥有不止一种文本符号表示应当算作符号多样性。不过，对于这一维度，我们还想考虑严格意义上的文本之外的表示法，因此不希望这成为唯一相关的问题。理想情况下，应当合理加权，避免把某个狭窄类别中多种多样的表示法误认为更广泛意义上的符号多样性。例如，我们希望体现出：尽管 UNIX 针对不同情境提供了种类繁多的语言，但它在符号多样性方面永远无法与拥有多种语言和多种图形表示法的系统相提并论。

2. 该系统是否使用图形用户界面元素？这是一个专门的非文本符号表示类别，许多示例系统都采用了这种表示。
3. 能否将数据视为树结构进行查看和编辑？树结构在编程中很常见，但通常会以某种方式作为文本来处理。我们的一些示例为这种常见的数据结构提供了图形符号表示，因此这也是它们与其他示例的区别之一。

4. 系统是否允许自由排列和调整数据项的大小？ 我们仍然觉得，Boxer 和电子表格体现了前面三个问题未涵盖的某些特性，也就是这一点。在各自将树渲染为嵌套方框、将内容呈现为单层网格的约束下，它们都支持符号表示上的变化，以适应用户的情境。这些系统本可以 选择让方框整齐排列，或让单元格大小完全一致，但它们允许这些元素变化，因此在符号多样性方面额外得分。

问题											
1							✔	✔			✔
2		✔	✔	✔	✔	✔	✔		✔		
3				✔		✔	✔			✔	
4					✔	✔					
总计	0	1	1	2	2	3	3	1	1	1	1

A.3 备注与未来工作

这项维度量化工作迫使我们深入探究，并为这些维度应如何定义制定更明确的标准。即使没有生成图这样的目标，我们也建议将其作为一项有益的练习。

有必要澄清我们在此所做工作的意义。必须注意，最终选定某个特定定义，并不意味着取代或废弃我们最初采用的维度一般性定性描述。显然，我们的过程中存在太多变异来源，因此不能将此处的结果视为最终的、客观的、唯一正确的维度定义，或类似的结论。每一种变异来源都为有意探究此问题的人士指明了未来工作方向：

量化目标。我们希望得到一些数值，以生成一张图，使其大致符合我们对若干示例系统的直观定位。换言之，我们试图让这些直觉以及维度本身都更加精确。另一种截然不同的方法是完全不设“锚点”，而是将特定定义所产生的任何答案都视为基准真值。然而，这就要求在提出问题和生成问题时提供更多细节。

问题生成。我们以非正式的方式生成问题，并在觉得问题数量足以区分各示例点的重要差异时停止了。不过，这方面的变化空间非常大，尽管以任何严谨的方式生成问题似乎都特别困难。也许，我们可以从一大组“系统运行时可以执行的操作”中选取自我可持续性问题，这样更容易对其进行参数化。同样，我们的符号多样性问题试图考虑几类符号表示；更完善的方法或许是统计广泛类别中的符号表示数量。

回答问题。我们通过达成共识，确定了对我们三人而言合理的答案。其他人可能不同意这些答案，而追溯分歧的根源，或许能为双方答案完全一致的其他问题带来启发。还可以通过以下方式获得有用信息：

让许多不同的人回答这些问题，并观察答案的差异有多大。

“Lisp”到底是什么？ 造成差异的最后一个主要来源，是我们为示例点指定的标签。在某些情况下（Boxer），确实只有一个系统；在另一些情况下（电子表格），有若干不同的产品，名称各异，但仍足够相似，可以合理地将它们作为同一事物来分析；还有一些情况（Lisp），我们将一系列相关系统视为设计空间中的一个整体点。如果有人认为这样做抹去了太多重要区别，这是可以理解的。在这种情况下，他们可以提议将其拆分为不同的系统或子系列，甚至可以建议如何将这些系列视为分布在各种子空间中的团块。

参考文献

[1] Megan Sapnar Ankerson.《*.com*设计：实用、社交与商业网络的崛起》. 纽约大学出版社，2018 年。ISBN：1479892904。[2] Antranig Basman. 构建软件（尚且）不是一门技艺。载于 Luke Church，编辑，《第*27*届编程心理学兴趣小组年会论文集，*PPIG 2016*，英国剑桥，*2016*年*9*月*7*日至*10*日》，第32页。编程心理学兴趣小组，2016年。访问于2023年2月9日。URL：https://ppig.org/papers/2016-ppig-27th-basman2/。[3] Antranig Basman. Infusion框架和组件，2021。访问于2023年2月9日。URL：https://fluidproject.org/infusion.html。[4] Antranig Basman、Luke Church、Clemens Klokmose 和 Colin B. D. Clark. 软件及其延续方式：将实时程序嵌入其周围世界。载于*PPIG*，2016年。[5] Antranig Basman、Clayton Lewis 和 Colin Clark. 开放作者原则：支持作者网络创建可外化的设计。载于 《*2018*年*ACM SIGPLAN*编程与软件新思想、新范式及反思国际研讨会论文集》，第29–43页，2018年。doi:10.1145/3276954.3276963。[6] Joshua Bloch. 如何设计优秀的 API 及其重要性，2007年。访问于2023年2月9日。URL：http://www.cs.bc.edu/~muller/teaching/cs102/s06/lib/pdf/api-design。[7] Marcel Borowski、Luke Murray、Rolf Bagge、Janus Bager Kristensen、Arvind Satyanarayan 和 Clemens Nylandsted Klokmose. Varv：作为声明式数据结构的可重新编程交互式软件。载于 计算机系统中的人为因素大会，CHI ’22，美国纽约州纽约，2022年。计算机协会。doi:10.1145/3491102.3502064。[8] Frederick P. Brooks. 贵族制、民主制与系统设计。载于 《人月神话：软件工程随笔》。Addison-Wesley，1995年。[9] Hasok Chang.《发明温度：测量与科学进步》。牛津大学出版社，牛津，2004年。ISBN：978-0-195-33738-9。

[10] Michael Alan Chang、Bredan Tschaen、Theophilus Benson 和 Laurent Van- bever. Chaos Monkey：通过系统性网络破坏提高软件定义网络可靠性。载于2015年ACM数据通信特别兴趣小组会议论文集，SIGCOMM '15，第371–372页，美国纽约州纽约，2015年。计算机协会。doi:10.1145/2785956.2790038。[11] Ellen Chisa. 介绍：错误轨道与 DB::get 匹配，2020年。访问于2023年2月9日。URL：https://youtu.be/NRMmy9ZzA-o。[12] Coda. Coda：汇集一切的文档。访问于2023年2月9日。URL：https://coda.io。[13] Evan Czaplicki. 什么是成功？，2018年。访问于2023年2月9日。URL：https：//www.youtube.com/watch?v=uGlzRt-FYto。[14] Jim des Rivières 和 John Wiegand. Eclipse：用于集成开发工具的平台。《IBM系统杂志》，43(2)：371–383，2004年。doi:10.1147/sj.432.0371。[15]Andrea di Sessa. 集成计算环境的原则性设计。人机交互，1(1)：1–47，1985年。doi:10.1207/ s15327051hci0101 1。_[16] Andrea di Sessa 和 Harold Abelson. Boxer：一种可重构的计算媒介。Commun. ACM，29(9)：859–868，1986年9月。doi:10.1145/6592.6595。[17] Jonathan Edwards. Subtext：揭示编程的简洁性。SIGPLAN 通讯，40(10)：505–518，2005年10月。doi:10.1145/1103845.1094851。[18] Jonathan Edwards、Stephen Kell、Tomas Petricek 和 Luke Church. 评估编程系统设计。载于第30届编程心理学兴趣小组年度研讨会，PPIG 2019，2019年。[19] Matthias Felleisen、Robert Bruce Findler、Matthew Flatt、Shriram Krishnamurthi、Eli Barzilay、Jay McCarthy 和 Sam Tobin-Hochstadt. 一种可编程的编程语言。Commun. ACM，61(3)：62–71，2018年2月。doi:10.1145/3127323。[20] Luciano Floridi、Nir Fresco 和 Giuseppe Primiero. 关于故障软件。Synthese，192(4)：1199–1220，2015年。doi:10.1007/s11229-014-0610-3。[21] John Foderaro. Lisp：导言。Commun. ACM，34(9)：27，1991年9月。doi:10.1145/114669.114670。[22] Martin Fowler. 领域特定语言。培生教育，2010年。ISBN：978-0-321-71294-3。[23] Nir Fresco 和 Giuseppe Primiero. 错误计算。《哲学与技术》，26(3)：253–272，2013年。doi:10.1007/s13347-013-0112-0。[24] Christopher Fry. 在已经满负荷的大脑上编程。Commun. ACM，40(4)：55–64，1997年4月。doi:10.1145/248448.248459。[25] Matthew Fuller. 软件研究：词典。麻省理工学院出版社，2008年。ISBN：978-0-262- 06274-9。[26] Richard P. Gabriel. Worse is Better，1991年。访问于2023年2月9日。URL：https：//www.dreamsongs.com/WorseIsBetter.html。[27] Richard P. Gabriel. 《作为设计者的设计》。载于第23届 ACM SIGPLAN面向对象编程系统、语言与应用会议论文集，

OOPSLA '08，第617–632页，纽约，纽约州，美国，2008年。计算机协会。doi:10.1145/1449764.1449813.[28] Richard P. Gabriel。《编程语言革命的结构》。载于*ACM* 编程与软件新思想、新范式与反思国际研讨会论文集，Onward! 2012，第195–214页，纽约，纽约州，美国，2012年。计算机协会。doi:10.1145/2384592.2384611.[29] Erich Gamma、Richard Helm、Ralph E. Johnson 和 John Vlissides。设计模式：可复用面向对象软件的基础。Addison-Wesley，马萨诸塞州雷丁，1995年。ISBN：978-0-201-63361-0。[30] Glide。Glide：无需编写代码即可创建应用和网站。访问日期：2023年2月9日。网址：https://www.glideapps.com.[31] John B. Goodenough。异常处理：问题与一种拟议的符号表示。*Commun.ACM*，18(12)：683–696，1975年。doi:10.1145/361227.361230.[32] James Gosling、Bill Joy、Guy Steele 和 Gilad Bracha。*Java* 语言规范。Addison-Wesley Professional，1996年。ISBN：978-0-321-24678-3。[33] Burton Grad。VisiCalc 的诞生与消亡。*IEEE* 计算史年刊，29(3)：20–31，2007年。doi:10.1109/MAHC.2007.4338439.[34] Thomas R. G. Green 和 Marian Petre。可视化编程环境的可用性分析："认知维度"框架。*Journal of Visual LanguagesandComputing*，7：131–174，1996年。doi:10.1006/jvlc.1996.0009.[35] Sumit Gulwani、William R Harris 和 Rishabh Singh。利用示例操作电子表格数据。*Communications of the ACM*，55(8)：97–105，2012年。doi:10.1145/2240236.2240260.[36] Christopher Hancock 和 Mitchel Resnick。实时编程与计算素养的核心理念。博士论文，麻省理工学院，2003年。访问日期：2023年2月9日。网址：https://dspace.mit.edu/handle/1721.1/61549.[37] Brian Hempel 和 Sam Lau。LIVE 研讨会，2021。访问日期：2023年2月9日。网址：https://liveprog.org/live-2021/.[38] Brian Hempel、Justin Lubin 和 Ravi Chugh。Sketch-n-Sketch：面向 SVG 的输出导向编程。载于第*32*届 *ACM* 用户界面软件与技术年会论文集，UIST '19，第281–292页，纽约，纽约州，美国，2019年。计算机协会。doi:10.1145/3332165.3347925.[39] Brian Hempel 和 Roly Perera。LIVE 研讨会，2020。访问日期：2023年2月9日。网址：https://liveprog.org/live-2020/.[40] Robert Hirschfeld、Hidehiko Masuhara 和 Kim Rose（编）。自维持系统研讨会，*S3 2010*，东京，日本，*2010*年*9*月*27*日至*28*日。ACM，2010年。doi:10.1145/1942793.[41] Robert Hirschfeld 和 Kim Rose（编）。自维持系统：第一届研讨会，*S3 2008*，波茨坦，德国，*2008*年*5*月*15*日至*16*日，修订精选论文，计算机科学讲义第5146卷。施普林格，2008年。doi:10.1007/978-3-540-89275-5。

[42] Daniel Ingalls。Smalltalk 背后的设计原则，1981年。访问日期：2023年2月9日。网址：https://archive.org/details/byte-magazine-1981-08/page/n299/mode/2up.[43] Alan Kay 和 Adele Goldberg。个人动态媒体。*Computer*，10(3)：31–41，1977年。doi:10.1109/C-M.1977.217672.[44] Stephen Kell。虚构的匹配模块：克服僵化软件构造的暴政。载于*OOPSLA Companion*，2009年。doi:10.1145/1639950.1640051.[45] Stephen Kell。操作系统：是否应该存在？载于第七届编程语言与操作系统研讨会论文集，PLOS '13，纽约，纽约州，美国，2013年。计算机协会。doi:10.1145/2525528.2525534.[46] Stephen Kell。有些语言注定属于 C：一种难以管理的语言何以经久不衰。载于*2017*年 *ACM SIGPLAN* 编程与软件新思想、新范式与反思国际研讨会论文集，Onward! 2017，第229–245页，纽约，纽约州，美国，2017年。计算机协会。doi:10.1145/3133850.3133867.[47] William Kent。单一事实的多种形式。载于论文集摘要。*COMP-CON* 春季 *89*。第三十四届 *IEEE* 计算机学会国际会议：智力杠杆，第438–443页，1989年。doi:10.1109/CMPCON.1989.301972.[48] Gregor Kiczales、Erik Hilsdale、Jim Hugunin、Mik Kersten、Jeffrey Palm 和 William G. Griswold。AspectJ 概述。Jørgen Lindskov Knudsen 编，载于*ECOOP 2001*——面向对象编程，第十五届欧洲会议，匈牙利布达佩斯，*2001*年*6*月*18*日至*22*日，论文集，计算机科学讲义第2072卷，第327–353页。施普林格，2001年。doi:10.1007/3-540-45337-7_18。[49] Ursula Klein。实验、模型、纸上工具：*19*世纪有机化学的文化。斯坦福大学出版社，斯坦福，加利福尼亚州，2003年。ISBN：978-0-804-74359-4。网址：http://www.sup.org/books/title/?id=1917。[50] Clemens N. Klokmose、James R. Eagan、Siemen Baader、Wendy Mackay 和 Michel Beaudouin-Lafon。Webstrates：可共享的动态媒体。载于第*28*届 *ACM* 用户界面软件与技术年会论文集，UIST '15，第280–290页，纽约，纽约州，美国，2015年。计算机协会。doi:10.1145/2807442.2807446.[51] Thomas Kluyver、Benjamin Ragan-Kelley、Fernando Pérez、Brian E. Granger、Matthias Bussonnier、Jonathan Frederic、Kyle Kelley、Jessica B. Hamrick、Jason Grout、Sylvain Corlay、Paul Ivanov、Damián Avila、Safia Abdalla、Carol Willing 和 Jupyter 开发团队。Jupyter 笔记本——用于可复现计算工作流的出版格式。载于国际电子出版会议，2016年。doi:10.3233/978-1-61499-649-1-87.[52] Donald Ervin Knuth。文学化编程。*The Computer Journal*，27(2)：97–111，1984年。[53] Jeffrey Kodosky。LabVIEW。*ACM* 编程语言会议录，4(HOPL)，2020年6月。doi:10.1145/3386328。

[54] 托马斯·S·库恩。《科学革命的结构》。芝加哥大学出版社，1970年。ISBN：0226458032。[55] Ranjitha Kumar 和 Michael Nebeling。UIST 2021——作者指南，2021年。访问日期：2023年2月9日。网址：https://uist.acm.org/uist2021/author-guide.html。[56] Dark 语言团队。Dark lang。访问日期：2023年2月9日。网址：https://darklang.com。[57] Steven Levy。《黑客：计算机革命的英雄》。Doubleday，美国，1984年。ISBN：0385191952。[58] Henry Lieberman。《如你所愿：示例编程》。Morgan Kaufmann，2001年。ISBN：978-1-558-60688-3。[59] Simon Marlow 和 Simon Peyton-Jones。《格拉斯哥 *Haskell* 编译器》。《开源应用程序架构》。知识共享，2012年。ISBN：978-1- 105-57181-7。网址：http://www.aosabook.org。[60] John McCarthy。《*LISP 1.5* 程序员手册》。麻省理工学院出版社，1962年。ISBN：0262130114。[61] Leo A. Meyerovich 和 Ariel S. Rabkin。Socio-PLT：编程语言采用原则。载于《*ACM* 编程与软件新思想、新范式及反思国际研讨会论文集》，Onward! 2012，第39–54页，纽约，纽约州，美国，2012年。计算机协会。doi:10.1145/2384592.2384597。[62] Stephen L. Michel。《*HyperCard*：完全参考指南》。Osborne McGraw-Hill，伯克利，1989年。ISBN：978-0-078-81430-3。[63] 微软。语言服务器协议。访问日期：2023年2月9日。网址：https://microsoft.github.io/language-server-protocol/。[64] Martin Monperrus。反脆弱软件原则。载于《首届编程艺术、科学与工程国际会议论文集》，Programming '17，纽约，纽约州，美国，2017年。计算机协会。doi:10.1145/3079368.3079412。[65] Ted H. Nelson。复杂信息处理：一种用于复杂、多变及不确定事物的文件结构。载于《*1965* 年第 *20* 届全国会议论文集》，ACM '65，第84–100页，纽约，纽约州，美国，1965年。计算机协会。doi:10.1145/800197.806036。[66] James Noble 和 Robert Biddle。关于后现代编程的笔记。*SIGPLAN* 通讯，39(12)：40–56，2004年12月。doi:10.1145/1052883.1052890。[67] Donald A. Norman。《日常事物的设计》。Basic Books, Inc.，美国，2002年。ISBN：978-0-465-05065-9。[68] Dan R. Olsen。评估用户界面系统研究。载于《第 *20* 届 *ACM* 用户界面软件与技术年度研讨会论文集》，UIST '07，第251–258页，纽约，纽约州，美国，2007年。计算机协会。doi:10.1145/1294211.1294256。[69] Cyrus Omar、Ian Voysey、Michael Hilton、Joshua Sunshine、Claire Le Goues、Jonathan Aldrich 和 Matthew A. Hammer。面向程序编辑器的语义基础。载于 Benjamin S. Lerner、Rastislav Bodík 和 Shriram Krish- namurthi 编，第二届编程语言进展峰会（*SNAPL*

2017），莱布尼茨国际信息学会议论文集（LIPIcs）第71卷，第11:1–11:12页，Dagstuhl，德国，2017年。Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik。访问日期：2023年2月9日。网址：http://drops.dagstuhl.de/opus/volltexte/ 2017/7127, doi:10.4230/LIPIcs.SNAPL.2017.11。[70] David Lorge Parnas。战略防御系统的软件方面。1985年。访问日期：2023年2月9日。网址：http://web.stanford.edu/class/cs99r/readings/parnas1. pdf。[71] Richard Pawson。Naked Objects。博士论文，Trinity College，都柏林大学，2004年。[72] Alan J. Perlis 和 Klaus Samelson。初步报告：国际代数语言。《ACM 通讯》，1(12)：8–22，1958年12月。doi:10.1145/377924.594925。[73] Tomas Petricek。软件中的错误计算：学会与错误共处。艺术、科学与工程编程，1(2)：14，2017年。doi:10.22152/programming-journal.org/2017/1/14。[74] Tomas Petricek 和 Joel Jakubovic。交互式编程系统的互补科学。载于《计算史与计算哲学》，2021年。[75] Ian Piumarta。可访问的递归理论基于语言的环境，2006年。访问日期：2023年2月9日。网址：http://www.vpri.org/pdf/rn2006001a_colaswp. pdf。[76] Eric S. Raymond 和 Bob Young。《大教堂与集市：一位意外的革命者对 Linux 和开源的思考》。O’Reilly，2001年。ISBN：978-0-596-00108-7。[77] James Reason。人为错误。剑桥大学出版社，1990年。ISBN：978-1- 139-07903-7。[78] repl.it。Replit：协作式浏览器集成开发环境。访问日期：2023年2月9日。网址：https://replit.com。[79] Barbara G. Ryder、Mary Lou Soffa 和 Margaret Burnett。软件工程研究对现代编程语言的影响。《ACM 软件工程与方法论汇刊》，14(4)：431–477，2005年10月。doi:10.1145/1101815.1101818。[80] Ben Shneiderman。直接操作：超越编程语言的一步。Computer，16(8)：57–69，1983年。doi:10.1109/MC.1983.1654471。[81] Kragen Javier Sitaker。编程语言背后的内存模型，2016年。访问日期：2023年2月9日。网址：http://canonical.org/~kragen/memory-models/。[82] Yannis Smaragdakis。下一范式编程语言：它们会是什么样子，又会带来哪些变化？载于《2019年 ACM SIGPLAN 编程与软件新思想、新范式及反思国际研讨会论文集》，Onward! 2019，第187–197页，纽约，纽约州，美国，2019年。计算机协会。doi:10.1145/3359591.3359739。[83] Brian Cantwell Smith。编程语言中的过程反射。博士论文，麻省理工学院，1982年。访问日期：2023年2月9日。网址：https://dspace.mit.edu/handle/1721.1/15961。[84] David Canfield Smith。Pygmalion：一种创意编程环境。1975年。[85] Guy L. Steele。Common LISP：语言（第2版）。Digital Press，美国，1990年。ISBN：1555580416。

[86] Guy L. Steele 和 Richard P. Gabriel。Lisp 的演变。载于第二届 *ACM SIGPLAN* 编程语言史会议，HOPL-II，第 231–270 页，纽约，纽约州，美国，1993年。计算机协会。doi:10.1145/154766.155373。[87] Steven L. Tanimoto。关于实时编程演变的观点。载于第一届实时编程国际研讨会论文集，LIVE '13，第 31–34 页。IEEE Press，2013年。[88] Philip Tchernavskij。设计与编程可塑软件。博士论文，巴黎-萨克雷大学，第580号信息与通信科学技术博士学院（STIC），2019年。[89] Warren Teitelman。*PILOT*：迈向人机共生。博士论文，1966年。[90] Bret Victor。可学习编程，2012年。访问日期：2023年2月9日。URL：http://worrydream.com/#!/LearnableProgramming。[91] Larry Wall。Perl：第一种后现代计算机语言，1999年。访问日期：2023年2月9日。URL：http://www.wall.org//~larry/pm.html。[92] Rudolf Winestock。Lisp 的诅咒，2011年。访问日期：2023年2月9日。URL：http://www.winestockwebdesign.com/Essays/Lisp_Curse.html。[93] Stephen Wolfram。*Mathematica*：计算机数学系统（第 *2* 版）。Addison Wesley Longman Publishing Co., Inc.，美国，1991年。ISBN：0201515075。[94] Melissa Rodriguez Zynda。第一个杀手级应用：电子表格的历史。交互，20(5)：68–72，2013年9月。doi:10.1145/2509224。

关于作者



Joel Jakubovic 是 Tomas 的博士最后一年学生，致力于克服编程中普遍存在的“过早承诺”问题。他即将完成的博士论文将探讨如何创建兼具自我可持续性与记号自由的编程系统。可发送电子邮件至jdj9@kent.ac.uk 联系他，并访问 programmingmadecomplicated.wordpress.com阅读他的博客。



Jonathan Edwards 是一位独立研究者，致力于大幅简化编程。他因 Sub- text 编程语言实验系列以及在alarmingdevelopment.org上撰写的博客而闻名。他曾在 MIT计算机科学与人工智能实验室和 CDG/HARC 担任研究员。他的推文账号是@jonathoda ， 可通过 jonathanmedwards@gmail.com联系他。



Tomas Petricek 是查理大学的助理教授。他致力于寻找更简单、更易于理解的编程思考方式。为此，他将编程系统与工具方面的技术工作，与科学史与科学哲学研究结合起来。他的作品可见于 tomasp.net ， 也可通过tomas@tomasp.net联系他。