第 1 章

语言原则

我们通常所说的“编程语言”实际上是个总体概念：从底层的编码到高层的构架，编程语言扮演了不同的角色。所以，对语言的评价，无法脱离这些具体的层次和角色。在很久以前，人们眼中的编程语言是很单纯的：程序员们用某种语言编写代码，然后编译连接，最后得到一个可执行程序。这其中，不必细致地分析，无需精巧的设计，只有代码的堆积。确实，用代码堆积出可执行程序是编程的最终目的，而且我们应该始终将这一目的铭记于心；但是，随着程序复杂性的增长，语言的角色显然变得不那么单纯了——许多分析和设计技术渐渐浮出水面，赋予了编程语言新的内涵。

分析和设计阶段是面向对象技术大有作为的领域，而面向对象语言则是为实现阶段提供支持。但很多时候，“设计”和“实现”缺乏共同语言：它们的很多基本概念都不一致，与开发环境也配合得不默契，甚至目标都不尽相同。人们不假思索地把问题多多的传统软件开发方法照搬到了面向对象世界；而且，这样的方法常常会得到管理层的青睐，因为从表面上看，传统方法显得很系统很正规：项目参与者各司其职——项目经理、团队领袖、分析师、设计师以及程序员等等。不幸的是，这种做法过于简单化了，也不足以把事情办好，反而往往导致整个项目组变得士气低落、缺乏创新。

那么，就让我们挥别传统方法，拥抱更好的更为理性的软件开发方法吧。分析、设计、实现以及项目组织，这些都应该相互补充、和谐地整合到面向对象之巨幅画卷才对。这样，我们才能更为经济地向市场提供软件产品；这样，我们的开发团队才会变得创意十足、激情飞扬。当然，“面向对象”也不是万能灵丹，我们并不指望它能解决我们所有的问题。毕竟，骏马也需好骑手来驾驭。

面向对象编程语言不仅需要提供对实现（implementation）层面的支持，还需要提供对分析和设计层面的支持。否则，又怎么能称之为“高级语言”？可惜，还是有许多人没有认识到这一点，而依然在坚持着设计与实现的分离。在他们眼里，设计必须要用独立于编程语言之外的语言来完成。可是，要知道，“高级语言”并不是汇编语言加上华丽的语法那么简单，C语言就是这样的——所以C不是高级语言；只有整合了设计层面和实现层面的概念与表示法的语言才是高级语言。遗憾的是，有些面向对象语言并没有做到这一点。

项目结构确定了，语言及编译器之外的工具，比如make，也是不可或缺的。但是，对这些工具的重新评估显示，这些工具目前的分工并不理想：离“优化”状态的距离尚远。首先，程序员不得不做额外的簿记工作[1]，而这些工作本来是可以让计算机自动完成的。其次，这些工具“分工有余，合作不力”，于是，事情变得复杂而缺乏灵活性。

这种不幸的状况不是凭空产生的。现实总非完美，技术总有制约。于是，语言的设计者不得不一再采取折衷措施，削平语言的棱角，使之可以套入当时技术之封装[2]；而不是围绕软件开发的问题（也就是编程语言要解决的问题）来设计语言。或许，“折衷哲学”在当时确实必要，但现在似乎不太合适了。特别是软件系统变得日益复杂，计算机用户们也逐渐成熟，他们自然会提出更多的要求。在今天的软件开发技术水平下，仍然残留着对过去技术所做的折衷妥协，显得那么多余和不合理。

C++语言是个有趣[3]的尝试：它力图将面向对象技术引入一种传统语言。Bjarne Stroustrup极富远见卓识地将这两种技术融合在了一起，居功至伟。他的“面向对象探险历程”很早就开始了。当Bjarne在“面向对象”的惊涛骇浪中沉浮时，这个领域中的许多问题尚未解决，甚至这些问题根本未被广泛认识到。所以，Bjarne不应成为众矢之的——因为，他已经为C++语言中的每一项功能深思熟虑，权衡再三。但是，现在回过头来看，我们可以发现，我们并不能将C++称为优雅的面向对象语言[4]，而且由于当时UNIX环境对链接器（Linker）缺乏支持，以至于C++的编译器也不得不受到连累，缺乏一些高级语言应有的特性。

C++所暴露的问题，当然是有解决之道的。C++一路走来，披荆斩棘，多年来对它的研究从未中止过。所以，我们已经知道问题何在，也知道了答案何在。既然如此，我们就应该发展并采用新的编程语言来解决这些问题。幸运的是，这样的语言已经存在——它们是工业级的，为商业项目而设计的；而非局限于象牙塔之中的学人之玩具。它们是否广为采用，那就得交由工业界来决定了。

---

1 译注：bookkeeping——如果你是老板，那么好理解了：你让秘书替你做的多半就是“bookkeeping”工作 。

2 译注：主要指用当时技术能写出该语言的编译器或者解释器。

3 译注：在美国，如果别人对你的成果的评价是“It's interesting!”（哦，那很有趣啊），那多半是一种比较含蓄的说法，其实评价不高；如果评价是“It's impressive!”，那么恭喜你，你的成果令人惊叹！

4 译注：很多较早的教科书都将C++作为面向对象编程语言的典范来讲授。实际上C++是多范型编程语言，同时支持基于过程、面向对象、泛型等多种编程方式。但本书作者的立场是，将“面向对象”的概念泛化，将“继承/多型”和“泛型”看作类型扩展的不同方式而融入面向对象技术，然后以此坐标来考察几种语言是否“优雅”（而非“纯粹”），却也自成体系。所以请热爱C++尤其是读过 *The Design and Evolution of C++* 一书的读者耐心看下去。本书不少观点和 *The Design and Evolution of C++* 是针锋相对的，但听听另一种声音（也是来自大师的声音），你终会获益匪浅。

Java 的发展很值得一提，它走了一条和 C++不同的道路：与 C 的严格兼容性并不被 Java 看重。在面向对象的世界里，C 语言的后裔可不止有 C++和 Java 两种。Brad Cox 的 Objective-C（主要用于 NeXT OpenStep 环境）也是从 C 衍生出的面向对象语言。在苹果公司的下一代操作系统——Rhapsody[1] 中，Objective-C 将成为占支配地位的编程语言。Objective-C 更像 SmallTalk，因为它是一种运行时动态绑定的语言。

Java 和 C++都继承了 C 的语法，C++更接近一些。类似于 C 的语法因简洁而受欢迎，但它并非最规范或可读性最佳的语法[2]。我们不应该只因“语法和 C 语言不类似”而拒绝一种语言，因为语法只是语言的一小部分。学一种语言的语法或许只要几天，但学会如何用该语言写出良好的程序或许需要几个月甚至几年。我们应该有这个胸怀来拥抱更为清晰的语法。

我们不应该只从技术角度（指语法和语义）来评价某种编程语言；还应该从“对软件开发的贡献”的角度来衡量。语言是用来交流的，如果团队内部担负不同任务的成员——从设计企业策略的管理者到得出测试结果的测试员，都能用一种语言（当然，是指编程语言）来顺畅交流，那么该语言就是成功的。因为这些人都与编程工作有关，所以编程语言理应让他们可以超越空间和时间的阻碍而自由交流。为何提到“时间的阻碍”？因为很少有一个程序员会在项目的全寿命期内始终参与。所以，后来者看到的往往只是他留下的“书面语言”。

用户永远是上帝。没有满足用户需求的软件就毫无价值。这倒不是说，最好的软件就一定拥有最多的用户；也不是说用户少的软件价值就少。在工业界，这样的逻辑已经被那些不懂得欣赏技术价值的人用得太多。在我们简化编程（哦，你问怎么简化？当然是靠改进既有的编程语言或者发明更好的编程语言啦☺）的努力中，我们必须时时注意保证我们生产的软件是有用且可用的。

1.1 程序设计

现在，编程和写规格书被看作是一回事——你的规格书就是我的程序。到最后，你大概可以拿个编译器直接去编译规格书，然后立即得到可运行的程序了。Morgan 将规格书和程序的差别一笔抹杀：“对我们而言它们都是程序” [Morgan 90] (1.2)。“编程”这个词，不仅仅是用来指实现，而应用来指分析—设计—实现的全过程。

---

1 译注：苹果新操作系统的开发一波三折，最终他们主导开发并采用了基于 FreeBSD 和 Mach 的开放内核 Darwin，以此为基础开发出 Mac OS X。

2 译注：类 C 语法简直就是最不规范的语法了——当年 Bjarne 为了写出可以让 Yacc（一个编译器辅助开发工具）接受的类 C 语法形式化定义，实在是绞尽脑汁。

Reade 对编程和语言做了这样的阐释：一种相当狭隘的看法是，一段程序就是为机器编写的一串指令序列[Reade 89] (1.1)。而我们想说的是，如果采取更为宽泛的观点，将程序看作是对值、属性、方法、问题、解的描述，那么我们将获益更多。这些描述提供了特定问题的解决方案，而计算机的角色不过是更为高效地处理这些描述。编程语言其实是一种约定，只有遵循了这种约定，我们的描述才能够由机器处理。

Reade 还将编程阐释为“关注焦点的分离” [Reade 89] (1.1.1) 。程序员同时要做几件事情： (1) 描述要计算什么； (2) 将计算序列分成小的步骤； (3) 计算时管好内存分配。

Reade 还说，在理想状态下，程序员应该将注意力集中在(1)上面，而不应为(2)(3)分心[Reade 89] (1.1.2)。显然，“行政工作”[1]很重要，但是，将这样的“行政工作”从主要任务(1)中分离，我们得到的结果可能更加可靠。让计算机来自动完成这些“行政工作”，编程会变得容易得多。

“关注焦点”的分离还带来了其他的好处。比方说，如果计算的序列和内存管理问题这些细枝末节不再出现在程序中，那么程序证明将变得可行得多。更进一步而言，由于在不同的计算机上合适的计算步骤并不相同，程序中描述“要计算什么”的部分也应与“怎样一步步计算”的细节分离。高度并行化处理机具有成千上百个分布在机内各处的处理器，并采用局部而非全局存储结构。当使用这样的计算机时，用“对于存储器中的数据对象施加一系列微小变化”来描述如何计算显然就不合时宜了[2]。

要让诸如内存管理之类的“行政工作”由计算机自动处理，就意味着设计和实现编程语言的人不得不认真对待它们。与程序员相比，语言实现者更有可能也更应该针对不同的计算

1 译注：administrative tasks，指（2）、（3）。

2 译注：对于并行处理机而言，合适的计算方法是，将变化在空间上而非时间上分散。例如，对于分散于不同局部存储器中的各个对象同步地施加变化。

机结构来采用不同的计算机制。

簿记工作

前面那些引自 Reade 的话很好地概括了本书用来分析和对比编程语言的原则。Reade 所说的“行政工作”也就是我说的“簿记工作”。“簿记工作”给软件开发增加了成本，而且降低了灵活性，灵活性的降低反过来又导致了成本进一步增高。C 和 C++常被批评为“含义模糊”，因为 C 专注于前面说的(2)、(3)，于是在 C 程序中因为充斥着为(2)、(3)而写的语句，以至于(1)要计算什么，反而显得不突出了。

高级语言描述的是“要计算什么”——也就是“问题域”。至于“如何计算”，这是低级的面向机器层面要处理的，是“实现域”。簿记工作的自动化意味着软件系统的正确性、兼容性、可移植性、高效性都将得到增强。簿记工作的由来，就是人们需要指明计算如何进行。而这暗含着移植性问题。

编程语言的一个最重要的原则就是让任务对于程序员显得不那么困难。而将簿记工作自动化，对达到这个目标的好处是不可低估的。我们要把高级语言和“具有花哨语法的汇编语言”分开。前者从程序员肩上接过了簿记工作的重担；后者则将系统底层的簿记工作压到了程序员的肩上。

我们可以这样来评判某种语言构造的利弊：看看这种构造是否移除了簿记工作。例如，在本书中我们将看到，泛型机制或与其相当的机制是很有价值的，因为它移除了“手工进行类型转换”的簿记工作（为了进行手工类型转换，你需要记得各个变量本来都是什么类型。而泛型机制将这种烦人的记忆工作交给了编译器。当然，好处还远不止于此）。Java 没有泛型机制[1]，所以 Java 程序员就比较辛苦了：比如，他们从 collection 中获取对象时，就不得不手工将其转回原类型。泛型是“说明性方法”[2]的一个例子。

说明性方法

高级语言接过簿记工作之重担的最明显方式，就是说明性方法。而低级语言则不然，它们多偏向“操作性方法”[3]，这使得它们看上去更像汇编语言。说明性方法使决策集中化，产生底层操作代码的任务则被交由编译器完成。而如果采用操作性方法，簿记工作就落到了程序员肩上了——他/她对某个特定实体进行了某种特定操作，一旦决策改变了，那么对该实体的所有该操作都需要改变；而不是像采用说明性方法那样，简单地修改一处声明然后重新编

1 译注：本书写作时没有，但现在有了。

2 译注：declarative approach。采用 declarative approach 设计理念的语言可以称为“说明性编程语言”（declarative programming language），它让程序员可以清晰地描述需要做什么，而不是巨细无遗地指明怎么做。Eiffel 就采取这样的设计哲学。再比如，SQL 是毫无争议的说明性语言（虽然不是通用编程语言）。例如，很典型的 SQL 语句：SELECT something FROM somewhere WHERE some-condition ——这很清晰地描述了要做什么，但根本没有具体指明怎么做。这种设计理念认为“具体怎么做”是语言本身及其支撑库和运行时系统的事。C++不是说明性语言。但是自从引入模板机制并逐步增强，且建立了诸如 STL 之类的库，C++也带有了“说明性”色彩，逐渐向“高级语言”过渡（这是 Bjarne Stroustrup 自己说的，见于 Bjarne Stroustrup 为 TCPL 中文版所写的前言及他的 D&E 一书）。不过 C++依然缺乏部分高级语言（或曰“说明性语言”）应有的机制。

3 译注：operator approach，按字面理解为“算符方法”。

译即可。C 及 C++就是主要支持操作性方法，所以程序员常常需要关注底层机制。高级语言则隐藏了这些实现细节，使得程序的开发和维护工作更具弹性。

我们将会看到，Eiffel 和 C++的最大不同之一是：Eiffel 更倾向于说明性方法，而不是提供一大堆用于实际操作的运算符。或许，有更多的运算符会让一种语言看上去更加强大，但说明性方法提供了同样强大的功能，而且是以更精细的形式，会带来更易理解的程序。Java 则位于 Eiffel 和 C++之间。

虽然 C 和 C++的语法与高级语言的语法相似，但并不能把它们看作高级语言，因为它们缺乏高级语言应有的特性，如我一再说的，免除程序员的簿记工作而让编译器来照料细节[1]。

高级语言最重要的特性就是免去程序员的簿记工作之苦，从而提高开发速度、增强程序的可维护性和灵活性。这一特性比“面向对象”本身更重要，它应该是所有现代编程方法的内在本性。

这一特性有助于帮助业界更为经济地生产软件。工业界应该告别目前那些代价高昂的技术，而向这样的理念靠拢。我们应该考虑我们要的是什么，评估我们面临的问题。面向对象技术提供了这些问题的一种解决方案。但是，面向对象技术的效力，则取决于实现的质量。

1.2 交流、抽象和精确性

任何语言的基本用途都是交流；撰写规格书也是一种交流，是和任务的其他相关人员交流，交流的内容当然就是要完成的任务。本书后面会提到契约表示法（contract notion），而契约（contract）的一部分就是描述要完成的任务是什么。在最底层，任务就是让计算机执行程序；再上面一层是程序的编译；更高层次的交流则是让其他人明白这项编程任务必须要达到什么目标。

在最底层，指令必须得到精确执行，但这里不涉及人类的理解，只有机器的操作，指令纯粹是写给机器看的。而在较高的层次，代码是否易于理解就很重要了，因为这是要给人看的。这也是为何开明的管理手段强调训练而非强制过程的原因。当然，这并不是说精确性就不重要了。恰恰相反，在较高的层次中精确仍是重中之重，因为如果丧失了精确性，那么其他一切努力都是白费的。许多项目失败了，就是因为早期的需求表述不够精确。

[1] 译注：C++中也有支持说明性方法的语言成分，比如模板就是。许多现代 C++代码中复杂而极具技巧性的模板技术之运用，可看作一种对“说明性方法”不太友好的语言进行 “说明性编程”的尝试。

不幸的是，经常是那些最没有编程经验的人工作在最高层，所以规格书常常缺乏应有的精确性和抽象性。正如 Dilbert 原理所说，工作效率最低的程序员被提拔到他们破坏性看上去最小的职位[Adams 96]。这一策略可不像人们一厢情愿所想那样是“制胜法宝”；相反，事实上它会带来最大的破坏。程序员团队面对着令人迷惑的任务，辛辛苦苦地揣测着规格书背后真实的意图，用代码一行一行地来阐述清楚那些含糊的字句；而与此同时，那些所谓的分析师们早已一路高歌向前，在新的项目里或新的公司里播撒着更多灾难的种子。

（确实如此——因为许多经理人员没有好好地读懂 Deming、De Marco 和 Lister 的著作，以及 Tom Peters 近来的作品，所以“工作环境及员工的态度会影响工作质量”这一真理在他们的圈子中已逐渐失传，或至少是被忽略了。或许，Scott Adams 的幽默，是这一真理唯一还能有影响的途径了）[Deming 82] [Latzko and Saunders 95] [DeMarco and Lister 87]。

在较高的层次，抽象有助于理解。抽象性和精确性对于较高层次的规格书都很重要。抽象并不意味着含糊，更不是放弃精确性。抽象是以某一视角为出发点，移去不相关细节。经过这样的抽象，你得到的是精确的规格书——精确地描述了相关的系统属性。事实上，抽象性和精确性是同一枚硬币的两面。

我曾听到过这样的说法：抽象是效率的天敌。这完全是对抽象的误解。正确的抽象实际上会提高效率。抽象并不是通过增加一个又一个的层次来隐藏底层的细节——尽管这种方法有其用武之地，比如使程序核心代码与底层较好地分隔从而达到较高的可移植性。

抽象是计算的基本概念。Aho 和 Ullman 曾经说过，“这个领域（指计算机科学）的一个重要部分就是，如何让编程更容易、让软件更可靠。但更为基本的是，计算机科学是一种抽象科学——建立问题的正确模型，设计合适的、对计算机可行的方法来解决问题。”更进一步地说，“就我们运用抽象的角度而言，它常常意味着简化——用可理解且可解的模型来替代复杂且充斥着各种细节的现实问题”[Aho 92] (第 1 章)。

举个众人皆知的例子吧。Harold Beck 设计的伦敦地铁线路图，就是将抽象性和精确性完美结合的典范。这张图很抽象，因为去除了一切和地铁不相关的城市细节。于是，这张图大小合适，也很易于阅读。这张图同时也非常精确——每条地铁线路，每个地铁站，每个交叉点，都如实标注。这里精确意味着站点“一个都不能少”，而且在线路上的次序不能颠倒。

许多其他的城市交通系统都遵循了与 Beck 的线路图同样的原则。采用这样的模型，乘客们能轻易地解决诸如 “我怎样才能从骑士桥到贝克街” 之类的问题。

事实证明 Beck 的抽象是受欢迎的，因为它能让乘客们高效地找到他们的乘车路线，高效地利用城市公交系统。这种高效性完全是得益于良好的抽象，所以抽象与效率从未为敌，只应为友。

1.3 表示法

编程语言应该能让项目成员相互交流观点、意图、决策。它应当提供一种形式化但可读[1]的表示法（notation）来支持对系统的自洽[2]描述，满足各种问题的需求。编程语言还需要提供支持自动跟踪管理项目的方法。这保证了满足项目需求的模块（类及函数）可以及时而经济地完成。编程语言可以帮助人们理清系统的设计、实现、扩展、纠错和优化时的思路[3]。

在需求调研、分析、设计各阶段，我们期待有形式化和半形式化的表示法。分析、设计、实现各阶段所用的表示法应该是互相补充的，而不是互相冲突的。目前，分析、设计和建模的表示法和实现的表示法实在是相差了十万八千里。当然，实现的表示法——也就是编程语言——对这一“代沟现象”很有责任：总体来说，它们太低层了。在这个问题上，设计师和程序员不得不相互折衷妥协，这样才能填补这一沟壑。现有的不少表示法为各阶段所提供的过渡之路是相当坎坷的。分析、设计、实现各阶段所用表示法之间的语义沟壑便成了臭虫错误之巢、繁文缛节之源。

好的编程语言应该是高层的需求分析和设计表示法的“面向实现之自然延伸”，这样才能带来需求分析、设计和实现更佳的一致性。面向对象技术很强调这一点。在面向对象技术中，抽象定义和具体实现可以分离，而它们用的是同一套表示法。

编程语言还提供了形式化地为系统撰写文档的表示法。程序源代码是系统的唯一可靠文档，所以一种编程语言应该显式地支持文档功能，而不是仅仅以注释的形式提供。对于所有语言而言，沟通的有效性都取决于写作者的技巧。好的程序员会要求语言有文档功能；他们也会要求语言的表示法清晰明了、易于学习。没学过编程的人也可以通过阅读程序来理解这个程序所描述的软件系统是个什么东西。毕竟，我们不能要求读报的人本人必须是专业的新闻记者，对吗？

---

1 译注：形式化（formal）是为编程自动化工具提供方便；可读（readable）则是为程序员提供方便。
2 译注：“自洽”意即某个理论体系或者数学模型的内在逻辑一致，不含悖论。
3 译注：在语言学家中也有持这样观点的学派，认为语言决定思考方式。

1.4 工具集成

编程语言的定义应该给集成的自动化工具——比如，浏览器、编辑器、调试器——留下发展余地。这样才能更好地支持软件开发。编译器也是个具有双重角色的家伙。第一个角色是负责为目标机器生成代码，而机器的任务是运行程序。编译器需要检查程序是否符合编程语言的语法和句法，以便理解这个程序并翻译成可执行的形式。编译器的第二个角色（也是更重要的角色）是检查程序员对于软件系统的表述是否合法、完整、一致——这意味着它进行语义检查，看看这个程序内部是否协调一致。产生一个自相矛盾的系统是毫无意义的。

在工具集成问题上，鱼和熊掌不能兼得：开发平台应该用同一套通用的工具支持多种编程语言（可能工具并不会和每种语言都配合得很好），还是偏向于某一种语言，还是应该让每种语言有它们自己的开发平台以及为它们量身订制的辅助工具（这就使得该开发平台难于适应其他语言）？

1.5 正确性

认定什么是不一致性、如何检查不一致性，常常会引起热烈争论。争论的焦点在于，可检测出来的不一致性并非和现实中的不一致性完全吻合。有两派相反的观点。一派认为，过度代偿[1]的语言束手束脚，我们应该信任程序员。另一派观点认为，程序员也是普通人，所以也会犯错误；但程序在运行时崩溃则是无法容忍的。

两种观点都有他们的道理，所以两派本不应该争论的如此激烈。现代编程语言应该这样设计：常见的错误能检测出来，但是不能让人觉得太束手束脚，也不能妨碍程序员使用正常手段解决问题。本节提供了一些图表来帮助读者理解正确的平衡是什么样的——鱼和熊掌如何兼得。

图 1-1 到 1-5 展示了部分选择折衷情况——代偿不足[2]、代偿过度及理想情况。图 1-1 给出了图解。我们可以看到四个类别：正确的软件、有不一致性的软件（会导致运行时出错）、对错误过度敏感的软件（会导致编译时虚假警告）、执行效率不高的软件（运行时进行了太多检查）。

1 译注：overcompensation，指替程序员做了过多工作。
2 译注：undercompensation，指将太多本应由语言承担的工作留给了程序员。

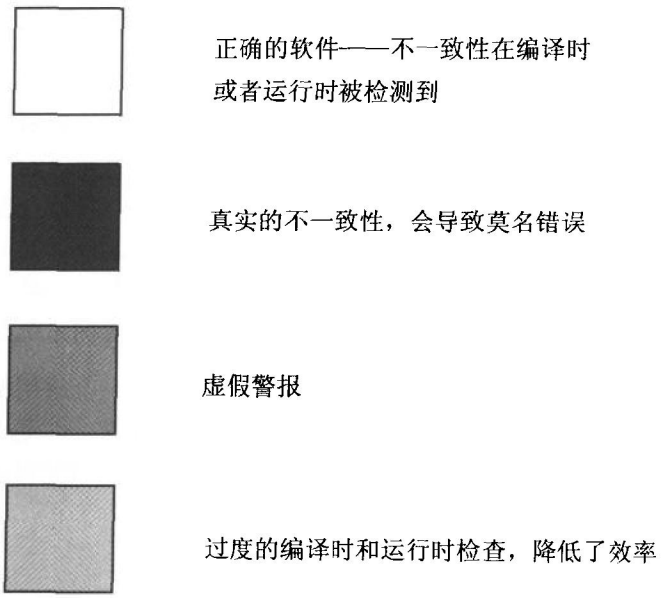

图 1-1 不一致性图解

在图 1-2 中，黑色盒子表示现实世界中的不一致性，它们必须被编译时或者运行时检测所捕获。在图 1-2 所示情境中，检查力度不够，所以不足以捕捉所有的不一致性，于是导致了运行时莫名其妙的错误，比如死机、结果错误，当然，在你运气特别好的情况下也可能平安无事。目前太多的软件开发是这样的：循环往复地编程-测试-修改，直到你发现好像一切都正常了——程序不再出错了——于是，可以交货了。我们将这种开发方式称为“hacking”。工业界的这种现状令人遗憾，我们应该尽早改变这种现状——当然，是通过采用更好的编程语言并摒弃这种“跟着感觉走，临场即兴发挥，直到幸运女神光顾”的开发手段。

本书要传达的观点是，语言和运行时环境应充分设计，以避免运行时莫名其妙的错误。正是这些错误导致了开发成本的上升和软件质量的下降。

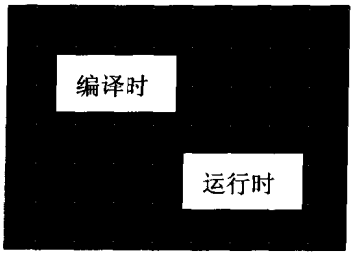

图 1-2 含糊的错误没有查出

某些人认为，编译器的检查带来了太多约束，而运行时检查则降低了效率，所以他们竭力鼓吹这一模型——因为程序员们是可以信任的，让他们来移除现实世界中其余的不一致性。尽管大部分程序员是尽责且可以信任的，但这种做法过于依赖运气了。用这种方法能够得到无错软件——只要程序员不首先引入不一致性。但是，随着软件系统规模和复杂性的增加，投入软件开发的人数也在增加，要做到这一点越来越难。现实不一致性常通过“编写—测试—

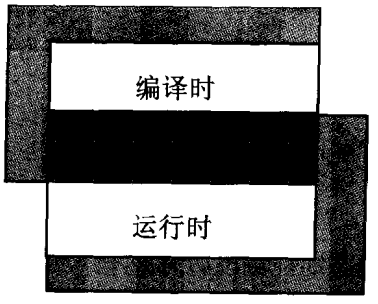

图 1-3 太多约束的编程语言

修改”的摸索式方法(也就是前文所述的“hacking”)来去除，这就导致了对测试的依赖性。有时候，公司甚至依赖客户来进行测试并提供反馈。确实，错误报告是客户和公司交流的重要途径，但是应把这看作软件质量的最后一道（也是代价最高的一道）防线。

优秀的程序员们在与各种环境的缺陷长期斗争中积累的经验、技术、模式，是一笔真正的财富。它们应被整合进现代的编程语言。

图 1-3 描述的语言对不一致性的检查过于严格了，超出了表示现实不一致性的方框。于是，我们将得到虚假的警告。编译器检查和移除了的不一致性，运行时支持环境还要重复处理，使运行时环境变得更加庞大而低效，语言也将拥有过多的约束。当然，用这样的语言开发软件，软件运行时你不会遇到各种莫名其妙的死机了；但是这种语言也妨碍了某些有用的计算手段的实施。Pascal 就经常因为限制太多（虽然有点不公正）而受到批判。

图 1-4 描述的情形更为糟糕，对于一些事实上并不存在的不一致性，编译器给出了虚假的警告——当然这也伴随着运行时过多的检测；而对于一些真正的不一致性，编译器却视而不见。

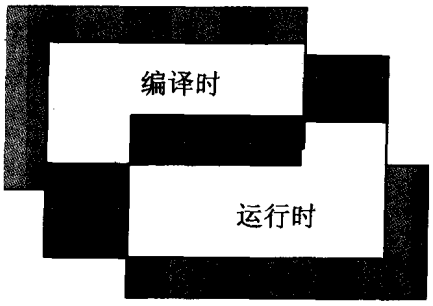

图 1-4 约束既过分又不足的语言

最理想的情况是，编译器静态地检测出所有的不一致性，而不产生虚假警报。如图 1-5 所示。

在这种理想的状态下，一个程序一旦编译通过，那么它就肯定是正确的，保证能够正常工作。更进一步地说，它将高效地运行，因为运行时检测已经很少了。尽管如此，在现有的技术水平下，我们不可能静态地检测出所有的错误，因为很大一类不一致性——例如，被零除、数组访问越界、本书 RTTI 部分讨论的类型检查和强制类型转换，只能在运行时才能检测出来。

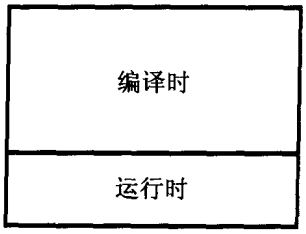

图 1-5 理想的语言

现在的目标是让可以检查的不一致性区域和现实世界中的不一致性区域重合，让运行时检查尽可能地少。这样做有两大优点：首先，运行时环境将更有可能无异常运行，结果是软件将更安全；其次，软件将更高效，因为不需要做很多的运行时检查。一种好的语言应该正确地将不一致性分类：哪些将在编译时检测，哪些将留到运行时检测。

为这个目标付出的代价是，这类语言的编译器必须做更多的工作，于是——所需耗费的时间将更长，甚至可能长的让人沮丧。但是，虽然开发耗时（实则编译耗时）增加了，但寻找导致无端失败之错误的时间以及与之相伴的挫折感大为减少，兴师动众大搞测试的成本也大幅度节省，所以这点代价也就显得微不足道了。Deming 曾说，我们应该停止对外部检测的依赖——它的一种形式是公司拥有专门的测试部门[Deming 82] (Point 3)。如果我们将 Deming 质量法则付诸实施，那么我们不得不对编译器期望更多。

这一分析显示了这样的结论：由于某些不一致性只能在运行时检测之现实，而这样的检测将导致异常，所以异常处理成为软件不可或缺的重要组成部分。不幸的是，在大多数编程语言中，异常处理并未受到足够关注。

本节自然地引出了“类型”的问题，因为类型健壮性对于去除运行时让人捉摸不透的错误有直接作用；或者说，只有一个类型系统可以避免运行时的奇怪错误时，才可称为健壮。Cardelli 对此给出了很好地解释[Cardelli 97]。下一节我们将讲述类型系统，并告诉大家一个良好设计的类型系统是如何来解决本节所描述的问题的。

1.6 类型

前一节我们讨论了编程正确性的条件。既然我们想确保程序在运行前就正确，那么“类型”这个话题我们就躲不过了。正如 Cardelli 曾说，“类型系统的基本作用在于避免程序在运行过程中的执行错误”[Cardelli97]，对一个程序来说，如果把所有的类型错误都检查出来，我们便可称之为类型安全；如果错误的检查全部发生在编译阶段而非运行阶段，我们称之为静态的类型安全。

要写出一个正确的程序，光靠检查语法是不够的，我们还需检查语义。有些语义是语言内置的，但大多数情况下语义是程序员就他们所开发的系统而指定的。

我们通过检查我们所使用的语言规范是否符合常理来完成所谓的语法检查。举例来说，“他喝着电脑，启动了一杯咖啡”，这句话从语法角度上讲一点错都没有，可是却连小学生都会嘲笑这种牛头不对马嘴的无稽之谈，这违反了我们关于电脑和咖啡的常识。用面向对象编程的术语来说，错误在于电脑这种“类型”没有“可喝”的属性。因此，在编写程序时我们也要尽量避免类似错误的发生，于是程序设计语言的类型系统便当仁不让地担当了检查这类错误的角色。类型定义了实体的属性与行为。

程序语言可以分为有类型和无类型两种；有类型的语言又可分为静态类型语言和动态类型语言两种。静态的类型检查在编译时便保证了只有合法的操作才能施与某一类型的对象，而动态的类型检查在运行阶段才检查程序是否有类型不一致之处。Smalltalk 是一种动态的类型语言，而不是无类型语言；Eiffel 是静态的类型语言。尽管动态的类型检查比较灵活，但它存在着弊端。由于在编译时编译器没有替程序员做类型检查，程序员不得不自己接过这付重担。程序测试工作就会变得很麻烦，为了保证程序所有的流程都经过检查我们将付出更大的代价。

C++是一种静态的类型语言，但我们有很多手段可以让程序具有无类型语言的灵活性，这意味着，我们可以强制让编译器忽略类型系统的强约束，而让一些特殊的程序通过编译。这就好比，有时我们不得不强迫那个可怜的人儿“喝”他的电脑，至于这之后此人是否需要被送往医院则是另一码事。有人认为，不允许这种偶尔的例外的语言是缺乏灵活性的。但是，正当的做法是，改变设计，让电脑具有可喝的属性☺。 我们不应该否决整个类型系统，因为类型系统正是灵活性所在之处，而非相反。定义及修改(类型)声明才是说明性编程(declarative programming)。相比之下，Eiffel 更偏向说明性，它的运算语法则相对简单；而 C++则提供了一大堆运算符。

定义复杂类型正是面向对象编程的核心概念。

在编程语言的发展过程中，最重要的里程碑也许就是引入了支持抽象数据类型(ADT)的特性[Bruce 96]。 这种特性允许编程者根据需要自己定义新的类型，而其操作如同语言的基本类型一样。编程者可以自己定义一种新类型，一大堆常量、变量和过程，并且可以禁止任何其他代码访问新类型的实现部分。特别是，只能通过给定的常数、函数、过程来访问这种类型的内部值。

OO 语言提供了两种特定方式来生成新的复杂类型：“对象能与其他类型以富有表现力和高效的方式结合（组合和继承），以定义新的更为复杂的类型” [Ege 96]。

面向对象编程最重要的特征也许是它的面向类型特性。事实上面向对象的未来应该是面向类型，或者说“面向对象”本来就应该是“面向类型”。面向类型的很重要的一点便是如何定义新类型，怎样利用已有的类型去派生更多的新类型（面向对象语言提供了并行不悖的继承和泛型这两种机制），怎样去指定类型之间的继承关系。在一个面向类型的系统中，我们需要一种类型表示法来描述基本类型，还需要一种类型运算法则来将已有的类型组合成新的类型。

一旦“面向类型”的思维框架和文化替代了“面向对象”的思维框架和文化，面向类型就会使得形式化的规格描述技术得到更为广泛的运用。这是面向对象分析（OOA） 和面向对象设计（OOD）方法论的未来。到那时，我们将会走出面向对象的宣传迷雾，“吹尽狂沙始得金”，面向对象技术也终将发展成熟，还其本来面目。到那时，软件工程最终成为真正意义上的工程。

Pascal 是一种早期的面向类型语言，尽管它不完美（事实上常常让严肃的程序员有挫折感），但受到 C 程序员的如此责难也是不公正的。事实上，直到 Bjarne Stroustrup 把一个复杂而精巧的类型系统引入 C，产生了 C++，C 家族才奠定了程序设计语言领域工业标准的地位。Pascal 的设计者 Niklaus Wirth 又设计了好多语言，包括现在的 Oberon，其核心概念是类型扩展。

本节的结论：正是多年前程序员们在反对 Pascal 的狂潮中不自觉地一并漠视了的类型，而不是伴随 C++而闻名遐尔的“对象”、“类”或者“继承”，才是真正使现实世界中的软件系统受益的机制——这不仅体现在它对类型正确性作出的贡献，还要归功于它避免了大量的运行时检查，从而使得系统性能得以优化。

1.7 灵活性、正确性和复用性

也许现在你已经相信，类型对于保证正确性实在是意义重大。类型带来了语义信息，但与此相伴的是潜在的危险性——它可能降低灵活性，以及复用的可能性。我们需要考虑，从无类型语言中我们获得了怎样的灵活性；以及，在移除无类型语言的“过度灵活性”的同时，我们能将多少灵活性植入类型系统。所以，本节要将灵活性、正确性和复用性放在一起讨论。

一种无类型的语言允许我们建立代码模式，并将任何实体施用于这一模式。比如，我们定义一个这样的模式，关联两个实体，返回另一实体：

$$x, y \bullet \textbf{if } x \circledR y \textbf{ then } a \textbf{ else } b$$

此处，我们并没有指定 x，y 是什么类型。这也就意味着，任何类型的 x 可以和任何类型的 y 相关联。® 本身是泛型的，可代表现实中的任何关系。这是一种很好的模式，许多更为复杂的模式也能按照这种方式来表述——你可以将任何类型的变量作为参数。因为这个函数适用于任何类型，你不必重写该模式，所以你拥有最大的复用性。因为涉及的实体无类型，所以我们拥有极大的灵活性。

但是，正如我们在前文中已讨论过的，这样的灵活性会带来一些荒谬的结果：比如“喝着电脑启动了一杯咖啡”。要举出一大把让模式显得毫无意义的实体和关系并不难。挑战在于，要将模式描述的可以预先防止那些可能导致系统错误的荒谬组合。如果从一个更为数学化的角度来看（实际上基于 λ 演算[1]——但我只作简单介绍，因为我可不想把读者吓跑☺），我们需要考虑一种特殊的关系。一个返回两个实体中较大值的函数可以采用前文的模式定义如下：

$$max \triangleq x, y \bullet \textbf{if } x \geq y \textbf{ then } x \textbf{ else } y$$

（“$\triangleq$”符号表示“以…命名”。也就是说，我把函数 $x, y \bullet \textbf{if } x \geq y \textbf{ then } x \textbf{ else } y$ 命名为 max。“$\triangleq$”符号还能读作“定义为”，但你不妨以另一种方式来理解：本来后面那一串东西没有

[1] 译注：λ 演算可以看作一种概念上的程序设计语言，用于评估语言功能和程序设计语言的研究和标注；它不产生可执行代码。

名字，现在我们给了它一个名字。在 λ 演算中，实际上并不给函数命名，只是将它们看作匿名实体。）好吧，我们知道，在宇宙中有许多实体是没有次序的，所以上述定义与其说具有极大的灵活性，不如说实际上限制了灵活性——因为它断言，任何东西都必须是有序的。例如，我们可以这样调用 max：

max(*pen*, *sword*)

这一表达式将展开成

{ *x*, *y* • **if** *x* ≥ *y* **then** *x* **else** *y* }(*pen*, *sword*)

并分别替换 x，y 为 pen 和 sword。于是，最终我们得到：

if *pen* ≥ *sword* **then** *pen* **else** *sword*

这个表达式有意义吗？笔和剑哪个更大一点？至少，在我们没有明确定义笔和剑的序列关系（比如，“笔的威力比剑大”）之前，这样的表达式毫无意义。关键在于，为了写好程序，我们必须精确定义这样的关系。这里我要做一点数学解释：x 和 y 称作绑定实体，那是因为它们是抽象概念，必须绑定到某一现实世界中的实体才有意义。这和第一个表达式中完全自由的实体 a 和 b 显然是不同的。

为了让 max 显得比较有意义，我们需要引入一些类型：

max ≜ *x*, *y*: *ORDERED* • **if** *x* ≥ *y* **then** *x* **else** *y*

但是，依然存在问题——因为我们可以将任何 ORDERED（有序）的类型关联到一起，哪怕它们不是同一类型。比如，我们用“已经书写过的字数”来给笔排序，而以“已经杀过的人数”来给剑排序，那么以上函数允许我们将笔和剑关联，因为它们都是 ORDERED 类型，但结果依然毫无意义。我们要么需要有一个规范的属性用来比较实体，要么需要更为严格的约束：实体不仅必须有序，而且必须同类。于是，我们最终得到：