面向对象语言的谱系

Peter Wegner

本文深入讨论了面向对象的程序设计语言。根据对象、类、继承性、数据抽象、强类型、并发性与持续性等语言特征，作为语言空间的设计量纲，讨论了语言的分类和层次关系。着重阐述了基于对象的语言、基于类的语言和面向对象的语言之间的联系与区别。

由于六十年代后期的软件危机，使得结构和简明性而不是表达能力成为语言设计的基础。软件工程学科应运而生，提出了一些结构程序设计与结构化设计方法论作为应用程序设计的基础。人们认识的重点从程序当作语句序列而转到了程序当作相互作用模块的集合。美国国防部为解决软件危机而开发的Ada语言支持了包括函数、过程、程序包、任务与类属程序单元在内的各种模块。

第一个面向对象的语言是六十年代中期开发的Simula，其特征是类的概念。它的实例由操作，协同程序和子类的集合组成，其中操作带局部状态，协同程序通过"resume"操作模拟并行执行，子类继承父类的操作与状态。

本文是作者对自己在OOPLA '87会议上的论文[1]所描述的概念作了增改而写成的。阐明了面向对象的程序设计的基本概念，并分析了传统的面向对象的顺序式语言的设计方法和适合整个面向对象程序设计的并发持续式(concurrent and persistent)语言的设计方法。

周使用电视录像带来编制知识获取过程的资料、特别是捕获由小组发现的软件攻坚技术。我们使用了电视录像带来回顾知识工程会议，研究攻坚实验，和教育其他工作人员掌握攻坚技术。此外，在第四周结束时，我们用电视录像带录制原型的演示情况以完成我们的视频资料工作。

软件攻坚术是一种迅速抓住棘手问题的有用工具。这种工具使管理部门可以考察，在一个组织调配时间和资金的主要开销之前，预计如何能够解决问题，或问题是否确实可以解决。软件攻坚术与速成原型的其它方法之区别在于：其时间结构大为缩短，只需3到4个月而不是1至2年。在这个实验项目的两个阶段中，总共花费的工作量为10个人月。

速成原型的一般结果是一个主要包括用户接口的系统外壳，而攻坚术将产生功能多得多的系统。在后继阶段后不久，在一次陆军通信会议上，我们演示了我们的原型。出席这次会议的陆军MSES设计者认为，我们的系统已达到设计者所需要的约75%的功能。

总之，我们发现，软件攻坚术是一种跳级启动原型设计项目的廉价而高效的技术。

参考文献(略)

〔中　原译自《COMPUTER》VOL.22 NO.5 1989 P39—48，王　心校〕

对象、类与继承性

说一个语言是基于对象的，是指它支持对象作为语言特征。这里对象定义如下：对象具有一个操作集合和一个"记忆"操作结果的局部共享状态。对一对象实施某操作所返回的值取决于对象的状态以及操作的变元。对象的状态如同局部存贮器，为该对象上的操作所共享。特别是，以前已执行过的其它操作有可能影响一给定操作返回的值。对象可以学习经验，把积累下来的经验结果(其调用历史)存贮在其状态中。

语言支持对象是值得的，因为语言的对象可以直接模拟现实世界中的客体，如银行、人、船只等。然而许多实际的面向对象的语言强加了其它一些语言需求，所以你应能分类与组织这类语言中的对象。特别是，它们要求对象属于某个类，而类又支持继承性。

类是一种模板，它可以用"new"或"create"操作来建立对象。同一类的对象具有共同的操作，从而也具有相同的行为。类具有一个或多个接口，指明通过它们操作是可存取的。"类体"指明实现类接口中操作的代码。

一个类可以继承父类(或超类)的操作，而该类本身的操作又可以被子类继承。由操作"C new"建立的类C的对象以C作为其基类，并可使用其基类或父类中定义的操作。对单个父类的继承叫单一继承，对多个父类的继承叫多重继承。

对象的语义

对象概念出自不同的语言有根本不同的表示(图1)。函数式对象产生于诸如OBJ2等的面向对象的函数式语言以及诸如Vulcan等的面向对象的逻辑型语言。其接口类似于对象，但对象在各操作之间不继续存在同一性。用于状态变换的操作导致产生具有给定接口和新状态的新对象。

被动式对象(或称为"伺服式"对象)在同步消息或远程过程调用唤醒其操作时成为活动的。Smalltalk与C++中的传统对象就是这种形式，当我们不加限定地使用术语"对象"时指的就是这种对象。这样我们可以把Ada的程序包、Modula-2的模

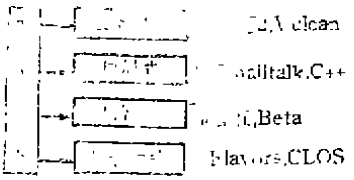

图1 不同语言中的对象概念

块、Simula与CLU的对象归入我们称之为对象的实体中。

主动式对象(或称"自治式"对象)，即使在其它对象没有调用它时也能自动执行。这类对象包括管程、强耦合并发对象与分布式并发对象。

基于槽的对象通过其实例变量(即槽，Slot)进行定义，如在Flavors与CLOS(公共Lisp对象系统)中就是如此。用DEFMETHOD操作可以动态地对基于槽的对象增加方法 (method)，DEFMETHOD操作的作用是指明要将方法加在其上的类(flavor)。

缺省定义不包括函数式对象与基于槽的对象，因为前者缺少对象同一性概念。而后者允许你动态地增加新方法。我们也许会把缺省定义进行拓广，使之包含基于槽的对象，因为Smalltalk也允许动态增加方法；然而，这种用法是很不常见的，因为其对象一般是用它们的方法在适当的位置上建立的。

对象可以用自动机模拟，自动机的状态表示对象的状态，而其输入符号表示对对象变元的操作。图2例示了一个具有操作$f_1, f_2, \cdots, f_n$的对象。

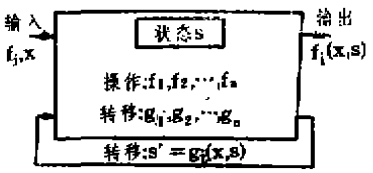

图2 一个具操作$f_1, f_2, \cdots, f_n$的对象

$\cdots, f_n$的对象。在状态s下对变元x的操作f_i产生输出$f_i(x, s)$与状态转移$s' = g_i(x, s)$。于是，操作符号f_i与输入带上的符号相关，内部动作与状态转移函数$g_i(x, s)$相关，而输出与操作$f_i(x, s)$相关。

我们可以用let表示法中的变式来描述对

象：

```
let x1=a1, x2=a2,… in
  f1(p1)=bcdy of f1
  f2(p2)=bcdy of f2
  …
endlet
```

变量x_1, x_2,…,x_n对应于实例变量，而p_1, p_2,…,p_n对应于操作(消息)f_1, f_2, …, f_n的参数。表示对象的let子句的语义有别于函数式程序设计中的let表示法：当一操作执行时，它可以修改实例变量。

实例变量x可以作为非局部变量出现在操作f_1, f_2, …, f_n中。于是，指定操作的这种面向对象的风格鼓励甚至要求在若干个操作中共享非局部变量。这种共享使得难以将某个类中的操作重用在其它类中。

面向对象的程序设计以牺牲各个操作的可重用性来获得效率与概念上的简明性。它力求通过封装与继承来达到对象与类的重用。

继承的性质

图3所示的继承层次关系定义了人与象为哺乳动物的子类，学生与女人为人的子类。(虚线下的)实例被组合成类，而(虚线上的)类归入其父类。于是，继承起组合类的作用，这十分类似于类起组合值的作用[2]。尽管诸如学生与女人等子类在现实世界中可能交迭，但它们在面向对象的语言中一般不能交迭，因为每个对象必须严格地属于一个基类。

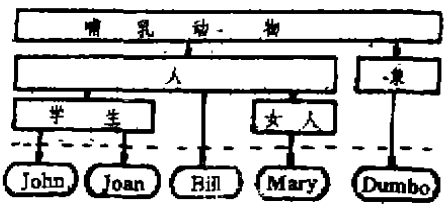

图3 继承的层次关系

继承性是概念上一个有效的组成原理，因为它可以吸取自然机制，诸如特化、抽象、近似与演化。这样，象是哺乳动物性质的特化，哺乳动物是象这一概念的抽象；哺乳动物的性质近似于象的性质；象是从早期哺乳动物进化而来的。类用于模拟概念，而类继承性使得对各概念之间层次关系的管理更容易。

在软件工程中，继承性不仅用于分类，而且也用于系统演化与增量修改。灵活地规定增量变化的继承能力是软件工程中一种极有价值的工具，但它需要一种从根本上比分类更有效的机制。例如，Smalltalk的继承机制的设计使你能既用之于分类，也能用之于模拟软件演化。

对增量修改的约束包括行为兼容性，特征(Signature)兼容性和名字兼容性以及消去[3]。行为兼容性要求被修改实体(不管是类还是对象)与其父实体的行为相容。同样，特征兼容性要求父实体的成分或属性的名字和类型的集合与被修改实体的相容。名字兼容性只要求父实体的成分名字相容。而消去允许你在被修改实体中消去父实体的属性。

这些形式的增量修改与各种基本模型相关，为不同的研究小组所研究。行为兼容性与代数模型相关[4]，特征兼容性与λ-演算模型相关[5]，名字兼容性与象Smalltalk这样的实现模型相关。消去与AI模型相关，它们常常涉及诸如这样的问题：企鹅虽然不会飞，但它们是鸟吗？

基于类与面向对象

我们可以根据对对象性质的约束来定义基于对象的语言的子类。诸如对象属于某个类与类支持继承性等要求是这种约束的例子。如果要求每一个对象都属于某个类，那么基于对象的语言也是基于类的语言(或传统语言)。如果继承机制可以增量定义类层次关系，那么基于类的语言也是面向对象的语言。

基于类的语言是基于对象的语言的真子集，而面向对象的语言又是基于类的语言的真子集(图4)。此外，这些语言类之所以有用，在于能根据语言特征区分现有基于类的语言并能把这种特征差异与语言所支撑的方法论

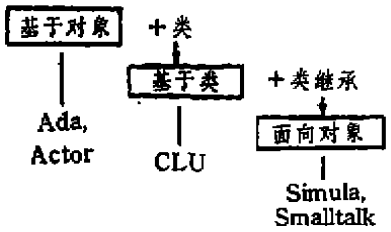

图4 从基于对象的语言到面向对象的语言

联系起来。

根据笔者的定义，Ada是基于对象的语言，它既不是基于类的语言也不是面向对象的语言，因为其对象(即程序包)不具有某个类(或类型)。CLU是基于类的，因为其cluster实际上就是类。cluster用作建立实例的模板并使实例成为"头等对象"，因为可以把它们赋给变量，作为参数传递、用作结构的成分。然而，CLU没有定义cluster间层次关系的继承机制，故它不是面向对象的语言。按笔者定义，Simula与Smalltalk是面向对象的语言。

图4可以看作一种用面向对象的技术对基于对象的语言进行分类的继承层次关系。你可以把基于类的语言看成继承了基于对象的语言的属性，把面向对象的语言看成继承了基于类的语言与基于对象的语言的属性。

我们简要考虑一下对象、类与继承性对程序设计方法论的影响。对象用于把操作及其所处理的数据组合在一起，提供面向数据的程序的设计原则。类用于管理对象的集合，允许把对象作为参数传送、赋给变量以及组织成结构。类继承用于组织类的集合，使类的层次关系可以描述应用领域。

诸如Ada等基于对象的语言支撑了对象的功能方面，但象程序库等不属于语言本身的机制必须实施对象管理，因为语言本身不支持程序库。基于类的语言提供了某种程度的对象管理但不作类管理。面向对象的语言可以在语言中实施对对象与类两者的管理，从而为应用的设计与实现提供统一的机制。由于面向对象的语言既支撑类层次关系的高层设计也支撑对象的低层实现，故它们属于"宽谱语言"。

数据抽象与强类型化

状态只能经由其操作访问的对象叫数据抽象。数据抽象向用户隐藏了对象的数据表示。例如，栈数据抽象(其数据只能经由Push与pop操作访问)就对用户隐藏了栈的表示(用表或数组)。这种数据抽象的类型叫抽象数据类型。

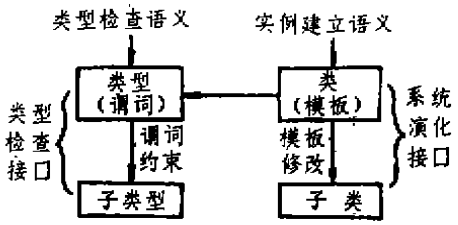

图5 类型与类

对象类型的概念与类的概念有很密切的关系。但是，类型是经类型检查激发的，可以用识别该类型的表达式的谓词定义，而类确定对象的集合，可以用建立对象的模板定义。类型具有类型检查语义，而类具有实例建立语义(图5)。

类型定义为谓词和类定义为模板，可以使我们根据谓词修改定义子类型，根据模板修改定义子类。子类型是根据下述约束定义的：该约束确定由其父谓词定义的集合的子集。子类是根据模板修改定义的，这种模板修改指完全修改(甚至消去)模板成分。

如果一个语言中所有表达式的类型能在编译时确定，则称它为静态类型语言。如果一个语言中所有表示值的表达式在编译时能从静态程序表示确定其类型兼容性，则称该语言是强类型语言。静态类型隐含着强类型，但强类型语言并不要求具有在编译时确定表达式的类型的能力，只要它们满足较弱的运算符/运算对象相容条件。

具有数据抽象与强类型的面向对象的语言属于更窄的一类语言，它们比我挑选作为面向对象的一类语言具有更强的构造性质。这类较窄的语言不包括Simula67与Smalltalk；前者由于其对象能访问它们的实例

变量，故其对象不是数据抽象；后者由于可以在执行的各个点上把不同类型的值赋给变量，故它不是强类型的。细心定义"面向对象"这一术语，既窄得把诸如Ada Modula与CLU等语言排除在外，也宽得把诸如Simula与Smalltalk包含在内。

一个面向对象的语言如果是强类型的并且要求所有对象都是数据抽象，则称为强类型的面向对象的语言。强类型与数据抽象的一个共同目标是强化对象的模块性，但它们在下面意义上讲也是无关的：对于不是数据抽象的对象，强类型也是可能的（如Simula-67），同样，如果没有强类型（事实上可以根本没有类型），数据抽象也是可能的（如Smalltalk）。

面向对象的语言应要求抽象与强类型吗？Smalltalk为进行动态汇集而有意避免了强类型，而在Flavors传统上的基于Lisp的面向对象的语言则有意避免了强类型与抽象。

Flavors甚至走得更远，它实际上没有把对象作为语言原语。它的一个概念是数据模板，在这种模板上可以加上操作。它没有指定操作，但可以用它作为操作的支点。从而，严格地说，Flavors风格的语言并不是基于对象的，但可以以基于对象或面向对象的方式将它用作一个基础。

引入数据抽象与强类型显然不一定有好处，一方面它要在结构与学科（discipline）间折衷，另一方面它也要在灵活性与效率间折衷。当你在设计过程早期进行特定抽象时，抽象就是有益的。但当你没有把握使精确抽象接近问题而希望试验抽象作为设计过程与原型制作过程的一部分时，就会觉得对抽象的约束太过分了。

AI应用或其它实验性应用的情况就是这样，它们涉及的是对构成一类问题的概念的理解而不涉及求解特定问题。基于Lisp的面向对象的系统打算用于这种应用，这类系统有意提供了非抽象对象以提高问题求解在概念上的灵活性。

非抽象的强类型语言有意义吗？尽管象Simula这样的语言可以说明，在不提供抽象的情况下也能提供强类型，但这也许是历史的偶然。非抽象对象很可能主要用在无类型形式体系中，这里由于没有抽象与类型而提高了概念上的灵活性。

类型的形式体系不利于实验性应用，以至于非抽象对象不再有用。然而，这仅仅是推测，更进一步的分析很可能会揭示出非抽象的强类型对象事实上在某种实验性应用中是有用的。

不考虑这些限制，总的来说，面向对象的强类型语言应成为应用程序设计尤其是整个程序设计的规范。面向对象的环境大概要支撑用于原型设计的Lisp风格的无类型程序设计以及用于传统应用程序设计的面向对象的强类型语言。此外，也应准备停止使用实验性原型代码，以便在准备用于产品程序设计时把它转换成强类型代码。

语言分类

根据越来越强的语言要求，我现在分出如下五类越来越小的语言：

1. 基于对象的语言，支持对象。
2. 基于类的语言，对象属于类。
3. 面向对象的语言，类支持继承性。
4. 面向对象的数据抽象语言，类支持信息隐藏。
5. 面向对象的强类型语言，类型可在编译时确定。

如果把并发性与持续性加到这组语言特征中，就可以把并发持续的面向对象的强类型语言类定义为支持并发性与持续性的面向对象的强类型语言[5]。无类的语言就是这样的语言，它们支持没有类的对象，并且为建立单类中的对象或建立执行操作时可能改变其类的对象提供灵活性。

无类语言又可进一步分为支持继承性的原型语言和支持并发性但不支持继承性的actor类语言。原型语言的开拓者是HenryLieberman[6]。actor类语言以放弃类与继承性的构造机制换取定义并发机制的灵活性[7]。

图6通过把语言特征视为语言设计空间的设计量纲例示了探讨语言设计方案的新方法。该例包含七种特征：对象、类、继承性、数据抽象、强类型、并发性与持续性。特征可以是二元的（即一个语言可以有之，也可以无之），这些特征可以有若干种供选择的实现，如对象、类与继承性即如此。

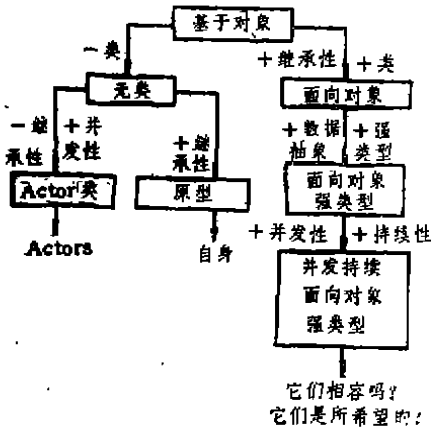

图6 基于对象的语言类

与七种语言特征的128个子集中每一个都可能有相对应的语言类，不过其中有些类要比其它类更令人感兴趣。笔者给每一个有趣的类取了一个名字，并考查了包含或排斥某特定特征的折衷方案。这种方法考查可能的面向对象的语言谱系是很有用的，因为它为探讨面向对象的语言的设计空间提供了系统化机制。

基于对象的并发性

基于对象的并发语言使用叫做进程的可并发执行的对象模拟客观世界。一个进程有一个可执行操作或入口点的接口和一个或多个控制线索（thread），线索不是活动的，就是被挂起的。基于进程的语言也是基于对象的语言，其对象（即进程）可并发执行。

进程中最小可执行元素是线索。线索是一种数据结构，装入处理机时就变成活动的。线索可以作为消息请求传递给进程，也可以在消息缓冲区中排队，直到一进程准备执行它们。线索在条件不适于其执行时亦可挂起，在适合条件出现时再活动。

进程可以按其线索的性质分类：顺序进程具有单控制线索；准并发进程至多有一个活动控制线索；并发进程有多个控制线索（图7）。

顺序进程（如Ada与NIL中的进程）一般有一个代码体和一个入口点接口，执行操作的消息可以在这里排队。调用操作（进入消息）在当前执行进程准备接收它之前必须等待。在当前执行进程准备接收它时，就进行会合，把进入的与活动的控制线索结合起来实施同步并且进行变元通信。然后控制线索再次分开，使调用进程与被调用进程再次并行执行。

准并发进程在等待要满足的条件时挂起控制线索的执行，当条件满足时再恢复执行。准并发进程与顺序进程的区别在于它们具有挂起线索的"条件队列"以及等待进入进程的线索的入口队列。仅在当前控制线索终止或挂起时，或当进入线索与活动线索经诸如会合等机制相结合时，进入线索才能成为活动的。管程是准并发进程的例子。

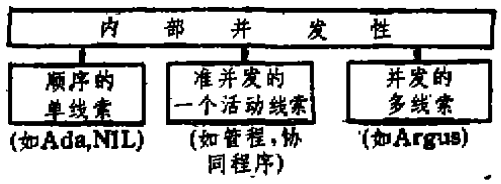

图7 内部进程并发性

并发进程对活动线索没有限制，调用操作可以自由地建立新线索。对临界区中共享数据的存取可能引起线索被挂起，直到安全地执行存取为止。进程中的并发性允许进行更细的粒状控制，从而使挂起从进程入口时延迟到临界区入口时。

并发语言CSP、Ada与NIL都具有顺序进程。基于管程的语言（如DP、ABCL/1与Orient 84K）具有准并发进程，Actor语言与Argus具有多个控制线索的并发进程（叫做警戒者）。

在概念建模与语言设计两个层次上可能

涉及关于进程是否应具有内部并发性的问题。从概念上讲，全并发进程可以更自然地模拟某些应用。然而，对设计和实现，顺序和准并发进程允许模块性与并发性单元是相同的，它们产生比并发进程简单得多的语言。

并发进程允许模块性单元包含多个并发性单元。因此，它们在语言与系统层对进程内与进程间的并发性要求有不相同的同步与通信机制[8]。

然而，并发进程要更为一致，你可以在进程内与进程间使用相同的并发原语。并发进程具有层次的（而不是平展的）进程结构。而且，它们允许更细的粒状并发性，在模拟要求这种并发性的现实世界时表达能力更强。

从顺序进程到准并发进程的过渡使得进程内线索的调度更灵活，不会引起因对数据结构简单访问所带来的互斥问题。然而，准并发进程在处理事务时也会出现互斥问题，因为在事务处理中间挂起线索会违背事务处理的完整性约束。

事务是一种原子工作单元，它可以要求与资源集暂时结合。由于事务表示了不可中断的暂时工作单元，故可将之视为"暂时模块"。准并发进程不会引起原子操作的互斥问题，但当你试图把事务的暂时模块性与传统的对象和进程的空间模块性结合起来时就会带来问题。

基于准并发进程的并发语言（如ABCL/1、Orient 84K)比基于并发顺序进程的语言更难以扩充以便进行事务处理。因此，在用准并发进程取代顺序进程时要在灵活性与扩充性之间折衷。

面向对象的并发系统必须能处理事务，从而也必须处理空间模块性（原子对象）与暂时模块性（原子动作）。

分布式进程

分布式进程是一种具有独立地址空间的进程，即，它不能直接访问其局部地址空间之外的任何资源，而只能经消息传送与外界通信。在作分布式进程的设计决策时涉及模块性单元(定义用户接口的单元)、并发性单元（表示单线索的单元）与命名单元（确定名字空间的单元）三者之间的相互制约（图8)。

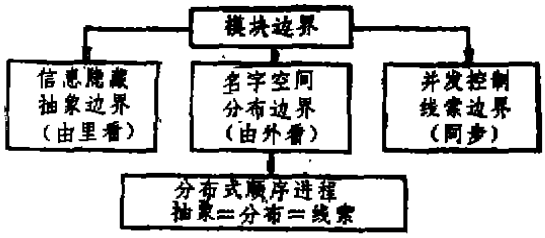

图8 模块性单元(左)、命名单元(中）与并发性单元(右)。

模块性单元确定模块的用户与模块内隐藏信息之间的接口。它确定模块中由外表看去的资源的可见性边界。模块性单元确定数据抽象的粒度，并发性单元确定线索同步的粒度，而命名单元（名字空间边界单元）则确定在程序任一点上环境中哪个名字集是已知的。

模块性单元、并发性单元与命名单元都相同的进程叫做分布式顺序进程。它们有优美的吸收力，因为你可以标出消息传送、互斥与事务之间的接口。但是，这样清楚地标识接口、并发性与名字空间是以牺牲概念上的灵活性与效率为代价的。之所以牺牲了灵活性是因为单元的共享必须与模块性单元和并发性单元具有相同的粒度，故模块或并发单元之间的共享是不可能的。之所以牺牲了效率是因为分布式成分之间的过渡需要很高的代价。

在分布式系统中可以分成静态互连分布式进程与动态互连分布式进程两大部分。前者中，每一进程与其环境的连接是在进程建立时确定的，并且在进程存活期间不能改变；后者中，语言命令可以在执行期间改变进程与其环境间的连接。

动态互连分布式进程中的端口是一些可

以赋给进程连接或通道的变量。慎重的做法是，使端口与类型相关，并且仅当端口值的类型和I/O方式与端口变量相容时才允许连接。输入端口可以想象为插座，而输出端口可以想象为必须与插座相吻合的插头。动态互连分布式进程可以用带有对应于通信通道的接线的可调插件模拟。

面向对象的持续性

持续性是用于确定数据保持时间的一种数据性质。在传统语言中，数据的存活期一般不能超出特定程序的存活期。有些数据（如局部说明的数据或过程参数）的存活期甚至更短。数据库所存数据的存活期（持续性）超出了各个程序的存活期（持续性）。在面向对象的语言中增加持续性，可以使之用作数据库实现的基础。

数据库可以视为一种具有特殊性质的存活期长的对象或进程，它可以由许多用户全局访问或共享。通常，从用户的观点访问是异步的，你可以把数据库想象成一种为异步用户请求服务的非终结进程。

异步访问可以直接由数据库进程处理，也可以由数据库服务程序处理，后者先组织用户请求，再把用户请求反馈给数据库。数据库本身可以是顺序进程（以串行方式处理请求）、准并发进程或全并发进程并且具有能强制数据访问互斥的锁。数据库进程需要如下专门特征：

·为支持持续性，需要一种很强的对象标识概念，它独立于对象选择中所用的关键码并持续存在于程序或项目中。

·需要一种查询语言以处理传统的数据库查询。这种查询可能涉及一种以上类型的对象，所产生的结果是对象的集合。关系数据库语言中的查询可以看作对聚集类型的select（选择）操作（所谓聚集类型指类型set（集合）或relation（关系））。它们具有形式：select (set, predicate)。

查询的复杂度与效率取决于谓词的性质。关系查询语言用受限原语集指定各个查询操作，其优化技术已得到广泛研究。面向对象的查询语言必须适应于更丰富的面向对象的规格说明，而相应的优化技术还未得到很好的理解。

面向对象的查询语言的一个问题是如何提高效率，不要使用户为面向对象的程序设计提供的灵活性付出效率上的代价。

·由于面向对象的数据库特别适合于管理演化系统，因此它们需要版本控制以及用于系统演化的其它工具。

·数据库应能规定约束，并能检查当数据库修改时有没有破坏约束。这可以用活动变量或触发子（trigger）来实现[9]。

·应提供多视图机制，当修改数据时要能自动更新所有视图。迟缓更新（Lazy update）当前未活动视图无疑是合适的。

八十年代的面向对象的程序设计正象七十年代的软件工程技术（如结构程序设计）一样变得越来越时髦，越来越重要。面向对象的程序设计这一术语已成为人们的口头语，但没有人知道它究竟是什么。在表征各种面向对象的程序设计时，我力求使这一术语的各种用法更准确。

我相信，如果Ada是现在设计的，它很可能会设计成一种面向对象的语言，而并发持续的面向对象的语言很可能成为九十年代程序设计的标准方法。面向对象的程序设计有着非常光明的前途。

参考文献（略）

〔徐宝文译自《BYTE》1989,3,P245—253
仲　源、声　钟校〕