OOA、OOD、OOP

JEDP 关注
2020.07.15 18:52:12 字数 10,001 阅读 526

OOA

Object-Oriented Analysis：面向对象分析方法

是在一个系统的开发过程中进行了系统业务调查以后，按照面向对象的思想来分析问题。OOA与结构化分析有较大的区别。OOA所强调的是在系统调查资料的基础上，针对OO方法所需要的素材进行的归类分析和整理，而不是对管理业务现状和方法的分析。

OOA（面向对象的分析）模型由5个层次（主题层、对象类层、结构层、属性层和服务层）和5个活动（标识对象类、标识结构、定义主题、定义属性和定义服务）组成。在这种方法中定义了两种对象类之间的结构，一种称为分类结构，一种称为组装结构。分类结构就是所谓的一般与特殊的关系。组装结构则反映了对象之间的整体与部分的关系。

OOA在定义属性的同时，要识别实例连接。实例连接是一个实例与另一个实例的映射关系。

OOA在定义服务的同时要识别消息连接。当一个对象需要向另一对象发送消息时，它们之间就存在消息连接。

OOA 中的5个层次和5个活动继续贯穿在OOD（画向对象的设计）过程中。OOD模型由4个部分组成。它们分别是设计问题域部分、设计人机交互部分、设计任务管理部分和设计数据管理部分。

一、OOA的主要原则。

（1）抽象：从许多事物中舍弃个别的、非本质的特征，抽取共同的、本质性的特征，就叫作抽象。抽象是形成概念的必须手段。

抽象原则有两方面的意义：第一，尽管问题域中的事物是很复杂的，但是分析员并不需要了解和描述它们的一切，只需要分析研究其中与系统目标有关的事物及其本质性特征。第二，通过舍弃个体事物在细节上的差异，抽取其共同特征而得到一批事物的抽象概念。

抽象是面向对象方法中使用最为广泛的原则。抽象原则包括过程抽象和数据抽象两个方面。

过程抽象是指，任何一个完成确定功能的操作序列，其使用者都可以把它看作一个单一的实体，尽管实际上它可能是由一系列更低级的操作完成的。

数据抽象是根据施加于数据之上的操作来定义数据类型，并限定数据的值只能由这些操作来修改和观察。数据抽象是OOA的核心原则。它强调把数据（属性）和操作（服务）结合为一个不可分的系统单位（即对象），对象的外部只需要知道它做什么，而不必知道它如何做。

（2）封装就是把对象的属性和服务结合为一个不可分的系统单位，并尽可能隐蔽对象的内部细节。

（3）继承：特殊类的对象拥有的其一般类的全部属性与服务，称作特殊类对一般类的继承。

式的定义。在特殊类中不再重复地定义一般类中已定义的东西，但是在语义上，特殊类却自动地、隐含地拥有它的一般类（以及所有更上层的一般类）中定义的全部属性和服务。继承原则的好处是：使系统模型比较简练也比较清晰。

（4）分类：就是把具有相同属性和服务的对象划分为一类，用类作为这些对象的抽象描述。分类原则实际上是抽象原则运用于对象描述时的一种表现形式。

（5）聚合：又称组装，其原则是：把一个复杂的事物看成若干比较简单的事物的组装体，从而简化对复杂事物的描述。

（6）关联：是人类思考问题时经常运用的思想方法：通过一个事物联想到另外的事物。能使人发生联想的原因是事物之间确实存在着某些联系。

（7）消息**通信**：这一原则要求对象之间只能通过消息进行**通信**，而不允许在对象之外直接地存取对象内部的属性。通过消息进行**通信**是由于封装原则而引起的。在OOA中要求用消息连接表示出对象之间的动态联系。

（8）粒度控制：一般来讲，人在面对一个复杂的问题域时，不可能在同一时刻既能纵观全局，又能洞察秋毫。因此需要控制自己的视野：考虑全局时，注意其大的组成部分，暂时不详察每一部分的具体的细节；考虑某部分的细节时则暂时撇开其余的部分。这就是粒度控制原则。

（9）行为分析：现实世界中事物的行为是复杂的。由大量的事物所构成的问题域中各种行为往往相互依赖、相互交织。

二、面向对象分析产生三种分析模型

1、对象模型:对用例模型进行分析,把系统分解成互相协作的分析类,通过类图/对象图描述对象/对象的属性/对象间的关系,是系统的静态模型

2、动态模型:描述系统的动态行为,通过时序图/协作图描述对象的交互,以揭示对象间如何协作来完成每个具体的用例,单个对象的状态变化/动态行为可以通过状态图来表达

3、功能模型(即用例模型à作为输入)。

三、OOA的主要优点

（1）加强了对问题域和系统责任的理解；

（2）改进与分析有关的各类人员之间的交流；

（3）对需求的变化具有较强的适应性；

（4）支持软件复用。

（5）贯穿软件生命周期全过程的一致性。

（6）实用性；

（7）有利于用户参与。

四、OOA方法的基本步骤

在用OOA具体地分析一个事物时，大致上遵循如下五个基本步骤：

包括如何在一个类中建立一个新对象的描述。

第二步，确定结构（structure）。结构是指问题域的复杂性和连接关系。类成员结构反映了泛化-特化关系，整体-部分结构反映整体和局部之间的关系。

第三步，确定主题（subject）。主题是指事物的总体概貌和总体分析模型。

第四步，确定属性（attribute）。属性就是数据元素，可用来描述对象或分类结构的实例，可在图中给出，并在对象的存储中指定。

第五步，确定方法（method）。方法是在收到消息后必须进行的一些处理方法：方法要在图中定义，并在对象的存储中指定。对于每个对象和结构来说，那些用来增加、修改、删除和选择一个方法本身都是隐含的（虽然它们是要在对象的存储中定义的，但并不在图上给出），而有些则是显示的。

OOD

面向对象设计（Object-Oriented Design，OOD）方法是OO方法中一个中间过渡环节。其主要作用是对OOA分析的结果作进一步的规范化整理，以便能够被OOP直接接受。

面向对象设计（OOD）是一种软件设计方法，是一种工程化规范。这是毫无疑问的。按照Bjarne Stroustrup的说法，面向对象的编程范式（paradigm）是[Stroustrup, 97]：

l 决定你要的类；

l 给每个类提供完整的一组操作；

l 明确地使用继承来表现共同点。

由这个定义，我们可以看出：OOD就是“根据需求决定所需的类、类的操作以及类之间关联的过程”。

OOD的目标是管理程序内部各部分的相互依赖。为了达到这个目标，OOD要求将程序分成块，每个块的规模应该小到可以管理的程度，然后分别将各个块隐藏在接口（interface）的后面，让它们只通过接口相互交流。比如说，如果用OOD的方法来设计一个服务器-客户端（client-server）应用，那么服务器和客户端之间不应该有直接的依赖，而是应该让服务器的接口和客户端的接口相互依赖。

这种依赖关系的转换使得系统的各部分具有了可复用性。还是拿上面那个例子来说，客户端就不必依赖于特定的服务器，所以就可以复用到其他的环境下。如果要复用某一个程序块，只要实现必须的接口就行了。

OOD是一种解决软件问题的设计范式（paradigm），一种抽象的范式。使用OOD这种设计范式，我们可以用对象（object）来表现问题领域（problem domain）的实体，每个对象都有相应的状态和行为。我们刚才说到：OOD是一种抽象的范式。抽象可以分成很多层次，从非常概括的到非常特殊的都有，而对象可能处于任何一个抽象层次上。另外，彼此不同但又互有关联的对象可以共同构成抽象：只要这些对象之间有相似性，就可以把它们当成同一类的对象来处理。

一、OOD背景知识

计算机硬件技术却在飞速发展。从几十年前神秘的庞然大物，到现在随身携带的移动芯片；从每秒数千次运算到每秒上百亿次运算。当软件开发者们还在寻找能让软件开发生产力提高一个数量级的“银弹”[Brooks, 95]时，硬件开发的生产力早已提升了百倍千倍。

为“组件（component）”。

类属是由被称为类（class）的实体组成的，类与类之间通过关联（relationship）结合在一起。一个类可以把大量的细节隐藏起来，只露出一个简单的接口，这正好符合人们喜欢抽象的心理。所以，这是一个非常伟大的概念，因为它给我们提供了封装和复用的基础，让我们可以从问题的角度来看问题，而不是从机器的角度来看问题。

软件的复用最初是从函数库和类库开始的，这两种复用形式实际上都是白箱复用。到90年代，开始有人开发并出售真正的黑箱软件模块：框架（framework）和控件（control）。框架和控件往往还受平台和语言的限制，现在软件技术的新潮流是用SOAP作为传输介质的Web Service，它可以使软件模块脱离平台和语言的束缚，实现更高程度的复用。但是想一想，其实Web Service也是面向对象，只不过是把类与类之间的关联用XML来描述而已[Li, 02]。

在过去的十多年里，面向对象技术对软件行业起到了极大的推动作用。在可以预测的将来，它仍将是软件设计的主要技术——至少我看不到有什么技术可以取代它的。

二、OOD到底从哪儿来?

有很多人都认为：OOD是对结构化设计（Structured Design，SD）的扩展，其实这是不对的。OOD的软件设计观念和SD完全不同。SD注重的是数据结构和处理数据结构的过程。而在OOD中，过程和数据结构都被对象隐藏起来，两者几乎是互不相关的。不过，追根溯源，OOD和SD有着非常深的渊源。

1967年前后，OOD和SD 的概念几乎同时诞生，它们分别以不同的方式来表现数据结构和算法。当时，围绕着这两个概念，很多科学家写了大量的论文。其中，由Dijkstra和 Hoare两人所写的一些论文讲到了“恰当的程序控制结构”这个话题，声称goto语句是有害的，应该用顺序、循环、分支这三种控制结构来构成整个程序流程。这些概念发展构成了结构化程序设计方法；而由Ole-Johan Dahl所写的另一些论文则主要讨论编程语言中的单位划分，其中的一种程序单位就是类，它已经拥有了面向对象程序设计的主要特征。

这两种概念立刻就分道扬镳了。在结构化这边的历史大家都很熟悉：NATO会议采纳了Dijkstra的思想，整个软件产业都同意goto语句的确是有害的，结构化方法、瀑布模型从70年代开始大行其道。同时，无数的科学家和软件工程师也帮助结构化方法不断发展完善，其中有很多今天足以使我们振聋发聩的名字，例如Constantine、Yourdon、DeMarco和Dijkstra。有很长一段时间，整个世界都相信：结构化方法就是拯救软件工业的 “银弹”。当然，时间最后证明了一切。

而此时，面向对象则在研究和教育领域缓慢发展。结构化程序设计几乎可以应用于任何编程语言之上，而面向对象程序设计则需要语言的支持[1]，这也妨碍了面向对象技术的发展。实际上，在60年代后期，支持面向对象特性的语言只有Simula-67这一种。到70年代，施乐帕洛阿尔托研究中心（PARC）的 Alan Key等人又发明了另一种基于面向对象方法的语言，那就是大名鼎鼎的Smalltalk。但是，直到80年代中期，Smalltalk和另外几种面向对象语言仍然只停留在实验室里。

到90年代，OOD突然就风靡了整个软件行业，这绝对是软件开发史上的一次革命。不过，登高才能望远，新事物总是站在旧事物的基础之上的。70年代和80年代的设计方法揭示出许多有价值的概念，谁都不能也不敢忽视它们，OOD也一样。

三、OOD和传统方法有什么区别?

还记得结构化设计方法吗？程序被划分成许多个模块，这些模块被组织成一个树型结构。这棵树的根就是主模块，叶子就是工具模块和最低级的功能模块。同时，这棵树也表示调用结构：每个模块都调用自己的直接下级模块，并被自己的直接上级模块调用。

靠上，概念的抽象层次就越高，也越接近问题领域；体系结构中位置越低，概念就越接近细节，与问题领域的关系就越少，而与解决方案领域的关系就越多。

但是，由于上方的模块需要调用下方的模块，所以这些上方的模块就依赖于下方的细节。换句话说，与问题领域相关的抽象要依赖于与问题领域无关的细节！这也就是说，当实现细节发生变化时，抽象也会受到影响。而且，如果我们想复用某一个抽象的话，就必须把它依赖的细节都一起拖过去。

而在OOD中，我们希望倒转这种依赖关系：我们创建的抽象不依赖于任何细节，而细节则高度依赖于上面的抽象。这种依赖关系的倒转正是OOD和传统技术之间根本的差异，也正是OOD思想的精华所在。

四、OOD步骤

细化重组类

细化和实现类间关系,明确其可见性

增加属性,指定属性的类型与可见性

分配职责,定义执行每个职责的方法

对消息驱动的系统,明确消息传递方式

利用设计模式进行局部设计

画出详细的类图与时序图

五、OOD设计过程中要展开的主要几项工作

（一）对象定义规格的求精过程

对于OOA所抽象出来的对象-&-类以及汇集的分析文档，OOD需要有一个根据设计要求整理和求精的过程，使之更能符合OOP的需要。这个整理和求精过程主要有两个方面：一是要根据面向对象的概念

模型整理分析所确定的对象结构、属性、方法等内容，改正错误的内容，删去不必要和重复的内容等。二是进行分类整理，以便于下一步数据库设计和程序处理模块设计的需要。整理的方法主要是进行归

类，对类一&一对象、属性、方法和结构、主题进行归类。

（二）数据模型和数据库设计

数据模型的设计需要确定类-&-对象属性的内容、消息连接的方式、系统访问、数据模型的方法等。最后每个对象实例的数据都必须落实到面向对象的库结构模型中。

（三）优化

OOD的优化设计过程是从另一个角度对分析结果和处理业务过程的整理归纳，优化包括对象和结构的优化、抽象、集成。

对象和结构的模块化表示OOD提供了一种范式，这种范式支持对类和结构的模块化。这种模块符合一般模块化所要求的所有特点，如信息隐蔽性好，内部聚合度强和模块之间耦合度弱等。

六、OO方法的特点和面临的问题

OO方法以对象为基础，利用特定的软件工具直接完成从对象客体的描述到软件结构之间的转换。这是OO方法最主要的特点和成就。OO方法的应用解决了传统结构化开发方法中客观世界描述工具与软

件结构的不一致性问题，缩短了开发周期，解决了从分析和设计到软件模块结构之间多次转换映射的繁杂过程，是一种很有发展前途的系统开发方法。

但是同原型方法一样,OO方法需要一定的软件基础支持才可以应用，另外在大型的MIS开发中如果不经自顶向下的整体划分，而是一开始就自底向上的采用OO 方法开发系统，同样也会造成系统结构不合理、各部分关系失调等问题。所以OO方法和结构化方法目前仍是两种在系统开发领域相互依存的、不可替代的方法。

七、OOD能给我带来什么?

问这个问题的人，脑子里通常是在想"OOD能解决所有的设计问题吗？"没有银弹。OOD也不是解决一切设计问题、避免软件危机、捍卫世界和平.....的银弹。OOD只是一种技术。但是，它是一种优秀的技术，它可以很好地解决目前的大多数软件设计问题——当然，这要求设计者有足够的能力。

OOD可能会让你头疼，因为要学会它、掌握它是很困难的；OOD甚至会让你失望，因为它也并不成熟、并不完美。OOD也会给你带来欣喜，它让你可以专注于设计，而不必操心那些细枝末节；OOD也会使你成为一个更好的设计师，它能提供给你很好的工具，让你能开发出更坚固、更可维护、更可复用的软件。

OOP

面向对象编程（Object Oriented Programming，OOP，面向对象程序设计）是一种计算机编程架构。OOP 的一条基本原则是计算机程序是由单个能够起到子程序作用的单元或对象组合而成。OOP 达到了软件工程的三个主要目标：重用性、灵活性和扩展性。为了实现整体运算，每个对象都能够接收信息、处理数据和向其它对象发送信息。OOP 主要有以下的概念和组件：

组件 – 数据和功能一起在运行着的计算机程序中形成的单元，组件在 OOP 计算机程序中是模块和结构化的基础。

抽象性 – 程序有能力忽略正在处理中信息的某些方面，即对信息主要方面关注的能力。

封装 – 也叫做信息封装：确保组件不会以不可预期的方式改变其它组件的内部状态；只有在那些提供了内部状态改变方法的组件中，才可以访问其内部状态。每类组件都提供了一个与其它组件联系的接口，并规定了其它组件进行调用的方法。

多态性 – 组件的引用和类集会涉及到其它许多不同类型的组件，而且引用组件所产生的结果得依据实际调用的类型。

继承性 – 允许在现存的组件基础上创建子类组件，这统一并增强了多态性和封装性。典型地来说就是用类来对组件进行分组，而且还可以定义新类为现存的类的扩展，这样就可以将类组织成树形或网状结构，这体现了动作的通用性。

由于抽象性、封装性、重用性以及便于使用等方面的原因，以组件为基础的编程在脚本语言中已经变得特别流行。Python 和 Ruby 是最近才出现的语言，在开发时完全采用了 OOP 的思想，而流行的 Perl 脚本语言从版本5开始也慢慢地加入了新的面向对象的功能组件。用组件代替"现实"上的实体成为 JavaScript（ECMAScript） 得以流行的原因，有论证表明对组件进行适当的组合就可以在英特网上代替 HTML 和 XML 的文档对象模型（DOM）。

在网络应用的增长中，一个很重要的部分是小型移动设备和特殊Internet设备的爆炸性增长。这些设备各有各的操作系统，或者只在某种特定的设备领域内有共同的操作系统。我们现在还可以一一列举出这些设备——家庭接入设备、蜂窝电话、电子报纸、PDA、自动网络设备等等。但是这些设备领域的数量和深入程度将会很快变得难以估量。我们都知道这个市场大得惊人，PC的兴起与之相比不过小菜一碟。因此在这些设备的应用程序市场上，竞争将会相当残酷。获胜的重要手段之一，就是尽快进入市场。开发人员需要优秀的工具，迅速高效地撰写和调试他们的软件。平台无关性也是制胜秘诀之一，它使得程序员能够开发出支持多种设备平台的软件。

我预期的另一个变化是，我们对于代码(Java)和数据(XML)协同型应用程序的开发能力将会不断提高。这种协同是开发强大应用程序的核心目标之一。我们从XML的迅速流行和ebXML规范的进展中，已经看到了这个趋势。ebXML是一个针对电子商务和国际贸易的，基于XML的开放式基础构架，由联合国贸易促进和电子商务中心(UN/CEFACT)与结构性信息标准推进组织(OASIS)共同开发。

我们能否期望出现一个真正的面向组件(component-oriented)的语言？它的创造者会是谁呢？

Stroustrup: 我怀疑，这个领域中之所以缺乏成果，正是因为人们——主要是那些非程序员们——对“组件”这个意义含糊的字眼寄予了太多的期望。这些人士梦想，有朝一日，组件会以某种方式把程序员赶出历史舞台。以后那些称职的“设计员”只需利用预先调整好的组件，把鼠标拖一拖放一放，就把系统组合出来。对于软件工具厂商来说，这种想法还有另一层意义，他们认为，到时候只有他们才保留有必要的技术，有能力编写这样的组件。

这种想法有一个最基本的谬误：这种组件很难获得广泛欢迎。一个单独的组件或框架(framework)，如果能够满足一个应用程序或者一个产业领域对所提出的大部分要求的话，对于其制造者来说就是划算的产品，而且技术上也不是很困难。可是该产业内的几个竞争者很快就会发现，如果所有人都采用这些组件，那么彼此之间的产品就会变得天下大同，没什么区别，他们将沦为简单的办事员，主要利润都将钻进那些组件/框架供应商的腰包里！

小“组件”很有用，不过产生不了预期的杠杆效应。中型的、更通用的组件非常有用，但是构造时需要非同寻常的弹性。

在C++中，我们综合运用不同共享形式的类体系(class hierarchies)，以及使用templates精心打造的接口，在这方面取得了一定的进展。我期待在这个领域取得一些有趣和有用的成果，不过我认为这种成果很可能是一种新的C++程序设计风格，而不是一种新的语言。

Lindholm: 编写面向组件的应用程序，好像更多的是个投资、设计和程序员管理方面的问题，而不是一个编程语言问题。当然某些语言在这方面具有先天优势，不过如果说有什么魔术般的新语言能够大大简化组件的编写难度，那纯粹是一种误导。

微软已经将全部赌注押在C#上，其他语言何去何从？

Stroustrup: C++在下一个十年里仍然将是一种主流语言。面对新的挑战，它会奋起应对。一个创造了那么多出色系统的语言，绝不会“坐视落花流水春去也”。

我希望微软认识到，它在C++(我指的是ISO标准C++)上有着巨大的利益，C++是它与IT世界内其他人之间的一座桥梁，是构造大型系统和嵌入式系统的有效工具，也是满足高性能需求的利器。其他语言，似乎更注重那些四平八稳的商用程序。

竞争

C#会不会获得广泛的接受，并且挤掉其他的语言？

言，语言的种类只会递增，也就是说，它们之间的关系是“有你有我”而不是“有你没我”。

对于一个新语言的接受程度，往往取决于其能力所及。Java技术被迅速接受，原因是多方面的，Internet和World Wide Web接口，在其他技术面前的挫折感，对于Java技术发展方向的全面影响能力，都是原因。另一个重要的原因是Java独立于厂商，这意味着在兼容产品面前可以从容选择。

C#是否会获得广泛接受？视情况而定。总的来说，那些对于平台无关性和厂商无关性漠不关心的程序员，可能会喜欢C#。那些跟微软平台捆在一起人当然可能想要寻找VB 和VC的一个出色的替代品。但是对于程序跨平台执行能力特别关注的程序员，将会坚守Java之类的语言。这种能力对于多重访问设备（multiple access devices）和分布式计算模型至关重要，而Java语言提供了一个标准的、独立于厂商运行时环境。

Stroustrup:C#的流行程度几乎完全取决于微软投入的资金多少。看上去C#的兴起肯定会牺牲掉其他一些语言的利益，但是事实上未必如此。Java的蓬勃发展并没有给C++带来衰败。C++的应用仍然在稳定增长（当然，已经不是爆炸性的增长了）。也许其他的语言也还能获得自己的一席之地。

不过，我实在看不出有什么必要再发明一种新的专有语言。特别是微软，既生VB，何需C#?

六、发展 vs. 革新

(Evolution vs. Revolution)

C++是一种发展型的语言，Java和C#似乎更像是革新型语言（它们是从头设计的）？什么时候，革新型的语言才是必需的呢?

Lindholm: Java技术并非凭空出世，反而更像是发展型的。Java所有的特性，在Java平台推出之前，都至少已经存在于另一种环境之中。Java的贡献在于，在众多的特性和权衡中，做出了合理的选择，使得产品既实用，又优雅。Java技术对于程序员的态度是：抚养，但不溺爱。

Stroustrup：从技术上讲，我并不认为Java和C#是什么“从头设计的”革新型语言。倘若Java是从技术原则出发，从头设计，大概就不会模仿C/C++那种丑陋和病态的语法了（不必惊讶，Stroustrup在很多场合表示过，C++采用C的语法形式，实在是迫于兼容性。他本人更偏爱Simula的语法——译者）。

我认为，只有当程序员们面对的问题发生了根本的变化的时候，或者当我们发现了全新的、极其优越的程序设计技术，又完全不能为现存语言所支持的时候，我们才需要全新的语言。问题是，我们恐怕永远也碰不到那些“根本”、“全新”的情况。

我以为，自从OOP问世以来，可称为“根本”的新型程序设计技术，唯有泛型程序设计（generic programming）和生成式程序设计（generative programming）技术，这两项技术主要是源于C++ templates技术的运用，也有一部分曾经被视为面向对象和函数式语言(functional languages)的次要成分，现在都变成正式、可用和可承受的技术了。我对于目前C++模板(template)程序设计的成果非常兴奋。例如，像POOMA, Blitz++和MTL等程序库，在很多地方改变了数值计算的方式。

C#的一个“卖点”，就是它们的简单性。现在Java是不是快失去这个卖点了?

Stroustrup：新语言总是宣称自己如何如何简单，对老语言的复杂性颇多非议。其实这种所谓的“简单性”，简单地说，就是不成熟性。语言的复杂性，是在解决现实世界中极为烦琐和特殊的复杂问题的过程中逐渐增加的。一个语言只要活的时间够长，总会有某些地方逐渐复杂

Lindholm：Java技术的的功能在增加，需要学习的东西也在增加。不过功能的增加并不一定带来复杂性的增加。Java技术的发展，并没有使学习曲线更加陡峭，只是让它继续向右方延展了。

标准

标准化语言和开放型语言各自的优点和缺点何在?

Lindholm：对于一个开放、不允许专有扩展、具有权威的强制性标准语言或者运行环境来说，不存在什么缺点。允许专有扩展就意味着允许厂商下套子绑架客户。特别重要的是，必须让整个平台，而不只是其中一部分完全标准化，才能杜绝厂商们利用高层次的专有API下套子。客户要求有选择厂商的自由，他们既要有创造性，又需要兼容性。

Stroustrup：对于一个语言，如C/C++来说，建立正式标准（如ISO标准）最大的好处，在于可以防止某一个厂商操纵这种语言，把它当成自己的摇钱树。多个厂商的竞争给用户带来的是较低的价位和较好的稳定性。

专有语言的好处，一是流行，二是便宜（不过等你被套牢了之后，情况就会起变化），三是对于商业性需求可以做出快速的反应。

标准化语言的特点之一是，它不能忽略特殊用户的需求。比如我在AT&T中所考虑的东西，其规模、可靠性和效率要求，跟那些普通厂商关注的大众软件相比，根本不可同日而语。那些公司很自然只关注主要的需求。

然而，多数大机构和身处前沿的公司，都有着特殊的需求。C++的设计是开放、灵活和高效的，能够满足我所能想象的任何需求。跟其他的现代语言相比，C++的家长式作风可谓少之又少，原因就在这。当然，不能赞赏这一点的人会诟病C++的“危险”。

拥有正式和开放标准的语言主要是为编程工具的使用者和客户服务的，而拥有专属“标准”的语言，主要是为厂商服务的