标准PASCAL的面向对象程序设计方法

Jonathan P. Jacky & J. Kalet

摘　要

本文讨论了用非面向对象程序设计语言进行面向对象的程序设计方法，并已成功地用于一个大型医用模拟程序。作者认为，面向对象的设计并不一定真正需要语言的支持，可以看作为一种设计方法学。这一观点不论对从事面向对象研究的人还是对从事应用程序设计的人都能有所启发和裨益，因为面向对象的设计方法已经越来越受到重视和欢迎，人们正在逐步采用这种方法进行软件开发。

近几年来，面向对象的程序设计已引起人们极大的热情。在这种程序设计中，程序基于的对象是类似于记录的数据结构。每个对象的类型，即类，与一系列特殊的操作有关。这些操作称为方法，类似于过程。当用消息调用对象时，方法即被实现。面向对象的程序设计的好处是由于对象与被模拟或计算的项的特性非常一致，因而程序设计较为容易；由于对象支持模块化和封装，因而程序错误较少；由于添加新的对象方便，因而程序易于扩展。

面向对象的程序设计通常与特定的程序设计语言有关。第一个这样的语言是Simula[1]，最有名的是Smalltalk[4]。最近，面向对象程序设计的支持已加入Lisp[10]，C[3,9]和Pascal的新的方言中。然而采用新的程序设计语言并不总是实用的，因为面向对象的方法并不真正需要语言的支持，它可视为一种设计方法学。基本原则是必须把程序中的每个数据项看作为某些对象的属性，并且只通过调用为该对象的类而定义的方法之一来访问。专门的语言能实现之，但在其它语言中也可以。有几种用于CLU和Pascal[8]，Ada[2]，甚至Fortran[5]语言的面向对象的方法学。我们的注意力在Pascal方法的开发，它已经成功地应用于一个癌症辐射治疗处理的大的(30000行)图形模拟程序[6]。

通常，这些系统表示患者的解剖模型和辐射源的排列，并且计算患者体内辐射剂量的分布。医务人员用这些系统寻求传递高辐射剂量只对肿瘤而不伤害健康组织的辐射源排列。典型地，用户用菜单选择射源，并用显示光标的移动把它们置于适当的

---

a) 从类中导出的每个对象可以备目录和文件引用，它们在I/O对象中的使用优先于File(文件)参数。

b)如果文件输入出错，则开始与用户对话。

5. 结　论

面向对象的方法是许多应用领域的一种极其自然的编程方法。特别是在CAD/CAM中更是如此，因为其中很自然地存在许多对象和分类。

本工作中，我们试图构造一个面向对象的系统，它十分灵活，足以很好地用于原型应用领域，这样的工作与日俱增，但并不十分要求效率。

Consult的优点有：1)灵活的类型；2)灵活而方便的数据存取；3)实时交互式数据库；4)开放型系统。

主要的缺点是：1)要求比表系统(LISP)多30%～40%的内存空间；2)运行时间长。

consult很容易引入现存的Prolog程序中，从而给出数据库的结构，同时得到简单而有力的用户界面。

参考文献(略)

〔彭　敏　黄素红译自《International Symposium for Young Computer Professionals》Aug. 21-23, 1989, Beijing, China〕

位置进行编辑处理。用户可以边观察计算出来的剂量分布边调整辐射源位置优化处理方案，剂量分布一般以附加在患者解剖模型剖面视图之上的分离剂量等高线图形式显示出来。我们所写的系统现正在几个诊所使用[6]。

对象和类

对象是Pascal记录的实例。每个对象的状态为记录中域所取的值的集合。对象的类用Pascal类型说明定义；类的名字为枚举的类型值。为了在Pascal强类型化之前进行面向对象的程序设计，重要的是不能用不同的记录类型表示不同的对象类，相反，所有对象都是相同基记录类型的实例；不同的的类通过定义变体记录类型而建立。类名在用于区分变体的特征域中出现。下面给出在我们的应用中定义名为object的类型(包括名为CONTOUR, BEAM和BLOCK的类，还有这里没有命名的一些其他类的样本Pascal说明。没详细讨论的项用省略号(…)表示，可以是用于存贮对象状态的其他类名或记录域。这些说明也表示了用于支持面向对象的程序设计机制。

```
class=(CONTOUR, BEAM, BLOCK, …);
object=record
  print-name: string;
  instance-number: integer;
  …
  case object class: class of
    CONTOUR: …
    BEAM: …
    BLOCK: …
    …
    … {其他类}
    …
  end;
end;   {对象记录说明}
```

对象的建立和访问

在大多数面向对象的系统中，对象的实例能随需要建立和废弃。实际上，这是许多种模拟的共同特性和我们在应用中之所以采用面向对象方法的一个重要因素。因此，我们的对象记录通过Pascal指针变量被动态地分配和访问。

对象实例是相对永久的，大多数在经过许多操作后仍趋于保持不变。在Pascal中，对一过程局部说明的变量当从该过程出来时便消失了。因此必须通过在主程序级说明的变量来访问对象实例。这样，对每个过程来说这些实例是全程的。幸运的是不必说明一个分离的全程指针变量来适应每个会出现的对象实例。相反，可以用树或表把对象实例相互链接起来，只须说明单个指针变量(树的根或表头)。然而，这的确需要存贮对象状态的记录也要包括对其他对象的链接信息。由于所有对象类都是相同基记录类型的变体，单个指针类型能链接不同对象的汇合。上面给出的忽略了变体和表示了链接的相同记录说明的另一种表示如下：

```
object-pointer=↑object;
object=record
  print-name: string;
  instance-number: integer;
  next:object-pointer;
  sub: object-pointer;
  case  object-class: class  of
    …
    …
end;   {对象记录说明}
```

链接为next和sub两个域，这一特定的实现给出了一个对象树，正巧能适合我们的应用。在我们的模拟中，许多实体是由单个"顶层"对象构成的复合对象和许多结构对象，即子对象。施于顶层对象然后是其所有对象的相同操作是典型的。这样，用顶层对象的链表表示模拟可能使每个顶层对象有一个子对象链表。

我们的应用不同于用具有语言支持面向对象系统的唯一特点是，树数据结构支持属于应用的模拟操作并提供对象管理的运行时支持。这两个功能在概念上是截然不同的，而且能作用于不同的数据结构，但出于效率和编码方便的缘故而合并到一起了。

在此方法学中树的使用并非主要问题，其他应用可用简单的链表或若干个队列。这些选择所共有的基本特点是通过一个或几个全程变量访问任意多的对象。

方法

一个类的所有方法都位于一单独的Pascal过程中，每个对象类都有这样一个过程。所有这样的过程具有形如invoke-<class name>的名字。这些过程的第一个参数是消息。消息是包含了要接收该消息的对象的标识符域的Pascal记录实例，以及接收对象应执行的方法的选择器。每个方法是Pascal case句中的分枝。方法选择器是pascal枚举类型的

值。

一些方法涉及单个对象实例，例如，EDIT允许用户改变对象的某些属性，DRAW在图形设备上显示对象。其他方法涉及到整个对象的类。SELECT产生一个类中所有可用对象的菜单并允许用口选择一个用于其后的操作。该方法要访问整个对象树，如同方法CREATE和DELETE一样，实际上完成树的处理。在我们所选定的设计方法中没有在过程中隐含引用全程变量，但采用了参数传递。因此，对象树的根必须传递到每个INVOKE过程。这里是一个用于对象类BEAM的样本过程。

```
type
  method-selector
    = (CREATE, SELECT, EDIT, DRAW,
    DELETE, …) ;
procedure
  invoke-beam (message; message-record;
              root-object: object-pointer; …);
begin
  case message.selector of
    CREATE;
    …new(message.object-id)…
    EDIT; …
    DRAW; …
    …
    …{其它对象选择器}
    …
    DELETE;
    …dispose(message.object-id)…
  end;
end;
```

面向对象风格的一个重要特点是相同对象类的所有方法都收集在一起。实际上把所希望进行的操作传给对象。这与过程程序设计风格是不一致的，那里对象作为参数传给过程。在过程风格中，每个操作要使用包含有关对象类转移的case语句的大过程。当我们加入新的对象类的时候，面向对象风格的优点就变得清楚了。不必把每个新的case转移加入每个过程，而是写一个单个的用于新类的新过程。

消息传递

对象并非用invoke过程直接调用，相反，有一个消息传递机制。每个消息是Pascal记录类型的实例，具有指示接收对象和要调用的方法的域。

```
message-record=record
  recipient: object-pointer;
  method: method-selector;
  …
end; {消息记录}
```

消息传递通过把消息记录放入后进先出消息栈实现。主程序仅为一个循环，它检查该栈顶部消息，确定要传送哪一个对象类，以及调用包括了该类方法的invoke过程。

```
program main;
begin
  …  {initialize stack; push initialization
      message on stack}
  repeat
    case stack↑.recipient↑.object-class of
      CONTOUR;
        invoke-contour(stack↑, …);
      BEAM;  invoke-beam(stack↑, …);
      BLOCR; invoke-block(stack↑, …);
      …
      …  {其他对象类}
      …
    end;
  until done;
end;
```

invoke过程通过向其他对象传送消息完成其许多工作。例如，方法EDIT主要地是构造消息，DRAW据此消息画出正在编辑的对象。EDIT把方法置于栈中，然后退出。接着，主程序检查此消息，并再次为该对象调用过程，这时转移到方法DRAW。DRAW所做的最后事情是从栈中移去它自己的消息，EDIT消息仍在那儿。因此，用户有机会再编辑和重画相同的对象。直到遇到其他方法为止。随后EDIT从栈中移去它自己的消息。如果一个对象有子对象，那么该对象的方法DRAW也为其每个子对象构造DRAW消息，并在刚要退出栈之前把它们全置入栈中（退出后移去它自己的消息）。这样就保证了当顶层对象移去时，复合对象的所有部件能重画。整个模拟能通过向所有的顶层对象发送DRAW消息而重画。当启动代码初始化栈和把为用户显示选项菜单的消息置于栈中时，即启动运行。

由于要经常重复以前完成过的操作，用栈的方法实现消息传递已被证明是非常适合我们的应用的。程序员必须牢记的是，消息将按它们被传送

（实际上就是压栈）的相反次序被处理，当一方法要移去它自己的消息时必须要进行跟踪。对我们的方法学而言栈不是主要问题，可以用队列或一些其他更好的数据结构来替代。

我们的确以称为send的过程形式给出一些语法成分，send构造消息记录，把它放入栈中，并写出之。send过程如下：

send(object-id, message-selector, …)；

除主程序中的case转移之外，在任何地方要调用对象，程序将用send而不是invoke过程。这是该技术的一个非常重要的特点，它用到称为同质异相的性质：不同的对象用完全相同的方式调用[3]。因此消息的发送者不必知道接收者的类。

由于不需修改已有代码就能添加对象类，这就提供了实用的优点。在我们的系统中一个典型的操作是在所能达到的范围内向每个对象发送相同的消息（例如，通过向每个对象发送DRAW而实现更新显示）。实现这一操作的代码是包含了一个send调用的循环。加入新对象类时它继续工作，仅需修改的代码（除对象记录说明和新的invoke过程）为在主程序case语句加入新的转移以传送新的invoke过程。

子类和继承

大多数面向对象的语言都提供了一种使用所谓继承的机制从老的类产生新类的方便的方法。程序员要指出新类是先前定义的称为超类的类的子类。语言运行时的系统确保子类的实例包括相同的数据项并执行与超类相同的方法。程序员也能用新的数据项或方法增加子类，或通过定义具有相同名字的新属性补偿那些超类。

我们的方法学仅在非常有限的意义上提供继承：所有的类都可认为是由名为object的记录类型说明所定义的原始超类的子类，每个类继承在该记录中不变部分的域所定义的属性，这些域是print-name，object-class，用于标识对象的instance-number和链接其他对象的指针next，sub。只访问这些域的代码，如象发现对象以及建立和删除对象时完成树数据结构操作的代码，对所有对象都一样工作得很好。然而，并没有说明一个类名来指定超类。因为方法位于特定的invoke过程内，所以也没有封装类属的支持代码。那些类属操作作为来自其他invoke过程的方法内的过程而被调用。

我们的类通过定义对象记录数据类型中的变体而产生。用十分相同的方式，可以通过嵌套任意附望深度的变体进一步详细说明每个类。这些变体可以认为是子类，但没有单独的invoke过程；相反地，是通过超类的invoke过程来处理。任何不涉及嵌套变体记录域的代码也同样可处理子类；对子类的特定操作用相同invoke过程的方法内的专门的代码来处理。

讨论

这里所给出的技术，在我们的应用中已被证明是相当成功的，而且比以前用过的方法有着重要的改进。较早些时候设计的辐射治疗规划系统[7]没用动态分配记录表示模拟数据，相反，而是采用静态分配的记录数组，每个对象类有着不同的记录类型（这样也就是不同的数组）。因为必须跟踪哪一个数组元素被合法数据所实际占用和若干个数组的哪一个元素与复合对象有关，结果非常麻烦。当抛弃该设计时，它由40000行源代码组成，而且从功能上看比现在的30000行系统要差得多。

我们还建立了一个原型，如同这里所描述的一样，它以动态分配的变体记录树表示数据，但用了传统的过程编码风格。开始时工作得不错，但随着对象类和操作的添加，变得难以控制。随着每个新加入的特性，我们发现在许多不同的地方要加入和修改过程调用和case转移。现在的消息传递机制是有意义的改进，因为它允许用相同的方法对要调用的不同类施加操作。

Liskov和Guttag描述过Pascal的面象对象程序设计方法学[8]，也是用动态分配记录表示对象。每个记录类型与一系列过程有关。但他们没有用变体记录作为处理不同对象的手段，也没有给出消息传递技术。

我们的技术的实现代价是有意义的。发送消息的开销包括执行主循环，两个case转移（选择invoke过程，然后选择invoke内的方法），两个过程调用（send和invoke），以及消息本身的汇集和堆栈。实际上，由于对象的"粘度"很大，这些耗费是微不足道的：大多数计算都出现在方法内。

在我们的应用程序中，不用面向对象的语言的代价是要明显作出对象间的连接和管理对象的代码。在最低级，可以用标准Pascal的New和Dispose过程建立和破坏对象，但仍要给出实现有关表操作的代码。有时，有必要搜索仅用其类和唯一的整数实例号标识的对象。这些服务会由面向对象语言的运行时系统隐含地提供。幸运的是，所需要的程序设计工作量不是很大。我们相信，我们的技术能

分析系统性能的一种方法：时间Petri网

林伟国　严隽永（海军工程学院）

摘　要

本文利用时间Petri网，讨论了分析系统性能的一般方法。在此基础上，进一步讨论了局域网中的CSMA/CD方式，并将此方法扩展到令牌总线方式上，为分析系统的物理特性和结构设计提供了一种方法。

1．时间Petri网

一般概念上的Petri网只对系统的逻辑特性进行分析，而不考虑系统的时间特性及性能。时间Petri网概念是七十年代提出的，它对系统的设计者来讲是相当重要的，因为它考虑了时间这个因素，可以形式化地讨论系统的一些性能，并可证明系统的安全性和活跃性[2]。

Sifakis提出并给定的时间Petri网具有如下的特性[4]：

时间Petri网(TPN)是一个六元组 $N=(P,T,\alpha,\beta,\tau,V)$，其中：

P：位置的集合，$P\neq 0$

T：变迁的集合，$T\neq 0$

α：$P\times T\longrightarrow I_N$(整数)，为向前相关函数

β：$P\times T\longrightarrow IN$(整数)，为向后相关函数

τ：$(\tau_1,\tau_2,\cdots,\tau_i,\cdots)$，为一实数递增序列，称为实数基

V：$P\times T\longrightarrow\tau$，使得$\forall(P,\tau_j)\in P\times\tau$：$V(P,\tau_i)=\tau_j\Rightarrow\tau_j>=\tau_i$

对于TPN也有相应的Petri网图，用圆圈表示位置，而用短线表示变迁。在Petri网图中，从位置p_s到变迁t_j之间存在一有向弧当且仅当$\alpha(p_s,t_j)=n(s,j)\neq 0$。而从$t_r$到$p_w$存在一有向弧当且仅当$\beta(p_w,t_j)=n(w,r)\neq 0$。标记M将位置P中是否存在标志映射为整数：$P\xrightarrow{M}IN$(整数)。下面是几条TPN规则：

a)在TPN中的标志呈两种情况之一：可行或不可行。初始时，每一位置p含有$M_0(p)$个可行标志。

b)变迁t是可发射的当且仅当连向t的每一个p_i，至少含有$\alpha(p_i,t)$个可行标志。

c)变迁t的发射在允许的情况下可以连续不断地发生。变迁t的发生从p_s中将移出$\alpha(p_s,t)$个可行标志，p_w中将接收到$\beta(p_w,t)$

很好地适合其他容易用比较少量的不同对象类定义的问题，许多模拟问题都有这些性质。

支持具有大量不断变化的对象类的研究工作不是我们的目标。因为不论何时要加入新的类或方法选择器，整个程序必须重新编译，这将是十分麻烦的，这些改变暗示着要重新定义全部数据类型。

这一方法学需要程序员注意该程序设计语言未提供的机制。Pascal根本不提供封装机制：由于必须通过全程变量访问对象，所以所有代码总是能访问所有的对象。实际上，我们发现在复合对象中不经消息传递机制，从顶层对象的域读取数据，子对象的方法是有用的。这在面向对象的语言中是不可能的。在我们的方法学中，这只是由程序员的喜好和斟酌来实现的。

（参改文献略）

〔沈国沛　译自《Communication of the ACM》，sept. 1987, Vol.30, No.9 p772—776〕