软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(9):2020−2035 [doi: 10.3724/SP.J.1001.2011.03846]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种动态角色模型及其实现机制*

唐祖锴[1,3], 彭智勇[2+], 任 毅[1,4], 崔晓军[1]

[1](武汉大学 软件工程国家重点实验室,湖北 武汉 430072)

[2](武汉大学 计算机学院,湖北 武汉 430072)

[3](武汉理工大学 计算机科学与技术学院,湖北 武汉 430063)

[4](通信指挥学院 网络管理中心,湖北 武汉 430010)

Dynamic Role Model and Its Implementation

TANG Zu-Kai[1,3], PENG Zhi-Yong[2+], REN Yi[1,4], CUI Xiao-Jun[1]

[1](State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China)

[2](Computer School, Wuhan University, Wuhan 430072, China)

[3](School of Computer Science and Technology, Wuhan University of Technology, Wuhan 430063, China)

[4](Network Management Center, Communication and Commanding Academy, Wuhan 430010, China)

+ Corresponding author: E-mail: peng@whu.edu.cn

Tang ZK, Peng ZY, Ren Y, Cui XJ. Dynamic role model and its implementation. *Journal of Software*, 2011,22(9):2020−2035. http://www.jos.org.cn/1000-9825/3846.htm

Abstract: There are limitations with most of the current role models. For example, the issues of role creation and attachment have to be handled explicitly in the source code: the navigation between role objects and source objects is unidirectional, and messages cannot be transferred bi-directionally, etc. Such limitations always result in the core business becoming tangled with the control logic for role objects. A dynamic role model, named DR, can provide the automatic creating and destroying mechanisms of role objects and can also provide the bi-directional navigation between role objects and source objects. These functionalities make up the core of the use of the role model, and the control logic for role objects is also transparent to users. The implementation of DR is centered around prepositve objects and delegation mechanisms not only resolves the complex hierarchy of roles problem, but the dynamic role model also solves the compatibility problem with traditional object-oriented systems.

Key words: role; dynamic role model; delegation; prepositve object; object deputy model

摘 要: 很多角色模型的设计和使用存在着一些限制,例如:角色对象的创建及其与源对象的绑定需要通过编码显式完成;角色对象与源对象之间的单向链接使得消息不能在它们之间互相转发等.这些限制使得角色模型的使用较为繁琐,在程序设计中往往会将系统的业务逻辑和对角色对象的控制逻辑混杂在一起.被称为 DR 的动态角色模型除了相关工作的基本功能外还提供了角色对象的自动创建及其与源对象之间的双向链接,使得角色对象的使用变得透明.所有这些功能的实现都基于一种简洁、统一的前置对象机制,它不但能较好地处理复杂角色体系,还能与

* 基金项目: 国家自然科学基金(90718027); 国家重点基础研究发展计划(973)(2007CB310806); 湖北省自然科学基金(2008CDA007)

收稿时间: 2009-06-25; 修改时间: 2010-03-03; jos 在线出版时间: 2010-11-01

现有的面向对象系统兼容.

关键词: 角色;动态角色模型;代理;前置对象;对象代理模型
中图法分类号: TP311 文献标识码: A

虽然面向对象程序设计方法(OOP)为提高软件模块化程度发挥了极大的作用,但其局限性也越来越多地受到研究人员及用户关注,其中最明显的问题就是由于数据封装性所导致的对象灵活性(flexibility)缺失.经典OOP 中的核心概念是“类”和“对象”,类作为属性和方法的模板可用于创建若干对象,且同一类的所有对象具有完全相同的行为特征.这种数据封装性显然限制了现实世界中对象多样性的事实,即使是同一类的对象,其行为也可能需要或多或少地存在差异.虽然可以通过继承及多态机制来为具有不同行为特征的同一大类对象生成相应子类,但另一个问题仍然存在.在 OOP 中,对象一旦创建就不能改变其所属的类,但现实世界中实体则可能随时转变其所属类型,甚至同时具有多个类型的行为特征,而这种情况难以在 OOP 系统中直观地建模.虽然有的系统提供了多继承机制,但希望在运行过程中动态改变对象类型或扩展对象原有行为也并不容易.也许有的系统会采用如下方案解决上述问题:在目标类中新建一个对象,并在销毁原对象之前将其中的基本信息拷贝到新对象中,从而完成其类型转换.且不论该方案在技术上是否可行,单从语义上考虑就存在不一致性,对象所代表的实体还是原来那个,但软件系统中对该实体的表示已转变为一个完全不同的新对象,这种情况可能会使系统的某些实现机制变得更复杂,或者引起一些不可预料的异常.

近几年,在程序设计领域出现了一批较成熟的动态 OOP 语言,如 Ruby[1],Python[2],Groovy[3]等.它们可以利用动态技术在程序运行过程中为类增加新的属性或方法,甚至也可以为单个对象在其类定义之外增加单独的行为.这类语言的动态特性大多基于 Mixin[4]技术,虽然可在一定程度上提高对象灵活性,但并不能从根本上解决前述问题.例如,虽然可以动态改变类定义,但这种改变会影响到该类的所有对象,很难只改变其中一部分对象的行为;虽然可以改变单个对象的行为,但如果需要同时针对批量对象进行改变则会变得非常繁琐;对象也仍然不能变换其所属类型,某类的对象也还是不可能成为另一个类的对象.

实际上,针对前述问题,更多的解决方案都关注于对角色模型(role model)的研究和设计上.角色模型通过对现实世界的抽象,将每个对象中变与不变的部分分开建模.对象中不变的部分就是该对象所属的类,这部分行为特征在对象的生命周期中都不会发生变化,例如对象的 oid 等;而对象中可能发生变化的部分则被定义为该对象的角色,不同的角色具有不同的行为特征,对象能够在其生命周期中与不同角色进行绑定,也可以同时绑定多个角色而改变或扩展该对象原有的行为.但是,由于角色模型通常建立在 OOP 模型的基础之上,其功能特性及实现机制大多会受到 OOP 的约束,从而使得角色模型的使用仍然存在一些限制.

本文首先简要介绍角色模型领域的相关研究工作,并提出我们在角色模型动态性方面的改进,其实现机制也将详细描述.最后讨论如何利用动态角色模型解决角色模型应用中的一些复杂机制以及相关问题和后续研究工作.

1 相关研究分析

关于角色模型的研究工作并不是一个新的领域.从基本出发点来看,所有研究工作都希望能够在语言模型的层面直接支持角色模型所需概念,这方面的尝试可以追溯到 SELF[5]语言以及 Delegation[6]机制.由于这些语言或机制的运行基础是基于原型(prototype-based 或 object-based)的 OOP 系统,虽然它们也能实现继承等机制,并且能够为角色模型运行环境提供较好的底层支持,但因它们不能与之后成为主流的基于类(class-based)的 OOP 系统很好地兼容而没有得到广泛的应用.

此后,很多关于角色模型的研究希望基于现有 OOP 系统进行实现.Fowler 在文献[7]中详细讨论了通过设计模式[8]实现角色模型功能的各种方案,由于不需要扩展现有 OOP 语言的语法结构,因而可以较方便地集成到现有系统中.但是,由于角色概念并不被 OOP 语言所直接支持,需要通过设计模式才能得到间接表达,而设计模式的严格结构可能对应用带来难度.如果想进一步实现一些高级功能,如角色链、多角色等机制,则还需要对原有

设计模式进行较大改动.这些都会提高应用难度,降低应用灵活性.另外一些研究工作则通过在 OOP 语言中增加语法结构或关键字的方式,使得从语法层面上能够直接表达角色模型相关概念,然后再通过预处理或翻译程序将其还原成标准的底层语言结构.这些研究工作从技术层面上看,大多还是通过对设计模式的语法封装,或利用底层语言的某些特殊功能来实现.这样的方式并不能从根本上提供角色模型所需的一些丰富功能,或者因为过于依赖某些语言的专有特性而无法达到广泛应用.

近几年中,在角色模型研究领域比较有代表性的工作包括 PowerJava[9],EpsilonJ[10],ObjectTeams[11]等.从表面上看,它们都是通过扩展 Java 语言,使角色概念能够以自然直观的语法结构得到表达,而在底层实现机制上则大都采用了 inner class 机制来提供支持.虽然这 3 种语言在编程风格上非常类似,但细节上仍然存在一些差异.对于 PowerJava 和 EpsilonJ 而言,它们都需要通过显式编码完成角色对象的创建及其与源对象之间的绑定.这种方式的好处在于,能够获得对角色对象的外部引用,甚至能够在其生命周期中改变所绑定的源对象,这对于需要保持角色持续性的应用场合是合理的.但其缺点则在于,必须在程序中手工维护角色对象,这在角色对象较多或角色体系结构较复杂的情况下容易造成逻辑失控,本文后续章节也将对此进行讨论.在 ObjectTeams 中则提供了角色对象的自动创建机制,提高了角色对象的自动化管理.通过其所实现的 lifting 或 lowering 机制,对象能够根据其应用环境自动在源对象和角色对象的功能之间进行切换.虽然用户无法获得对角色对象的显式引用,但也不必关心如何创建和绑定角色对象.但是,ObjectTeams 没有提供角色对象的销毁机制,除非删除角色类的定义,否则角色对象会一直依附于源对象上,并直到源对象销毁后才会一并销毁.另外,ObjectTeams 所提供的角色机制是以类为粒度的,一旦为某个类定义了角色类,则该类的所有对象都会具有相应角色对象,这也不利于满足需求的多样化和灵活性控制.ObjectTeams 的实现和使用也相对复杂,对于一些语法结构和功能机制则需要通过细致的学习才能掌握.

基于以上讨论,本文提出一种动态角色模型.它在基本角色模型的功能之上还具有更加灵活的动态特性,能够以对象为粒度控制角色对象的自动创建和销毁,能够提供角色对象与源对象之间的双向链接,能够对复杂的角色链或角色树结构提供良好支持.最重要的是,所有这些功能都是在一种简洁统一的机制下实现的,并且能够与现有的 OOP 系统完全兼容,这使得用户在 OOP 领域的长期技术积累仍然能够在用动态角色模型开发系统的过程中得到利用.

2 动态角色模型的基本功能

2.1 基于对象代理模型的设计

首先需要说明的是,本文中所讨论的动态角色模型,其功能基础来源于我们之前关于对象代理模型(object deputy model)[12]的相关研究工作.对象代理模型是一种在 OOP 系统中融入了代理机制(deputy mechanism)的数据模型,主要应用于面向对象数据库领域,为改善复杂数据的建模、存储、查询等提供了有效解决方案,增强了复杂数据的表现多样性和灵活性.在对象代理模型中,代理类(deputy class)以不同的代理语义(如特化、泛化、联合、分组等)与源类形成继承关系,能够以不同方式扩充源类的状态和行为.

本文不详细讨论对象代理模型的相关细节,但通过分析不难发现,角色模型中的若干概念都能直接映射为对象代理模型中的相关概念,例如,角色类(角色对象)对应于代理类(代理对象)、源对象与角色对象之间的消息转发对应于切换操作、多源角色对象对应于联合或分组代理类等等.这些特点使我们很容易想到,可以将其移植到程序设计领域,用对象代理模型的功能特征来提供角色模型所需功能,并且与普通角色模型相比,能够提供更多的动态特性.

但是,对象代理模型更加注重于数据库环境下复杂数据的表现多样性问题,而运用于程序设计领域的角色模型则更加注重程序运行过程中源对象与角色对象之间的消息传递机制,这在动态角色模型的设计和实现中更加重要.因此,本文在基于对象代理模型的主要功能机制设计动态角色模型的过程中专门开发了一种适合于程序设计领域的简洁统一的底层实现机制,使得所需的动态功能及消息传递过程都能正常执行.

为描述方便,本文所讨论的动态角色模型将被命名为 DR.下节中将首先通过一个基于图书销售系统的简

单例子描述 DR 的核心功能特征,其代码语法与 Java 语言非常相似.由于采用了专门的底层实现机制,因而甚至无须扩充现有 Java 的语法结构就能利用 DR 进行程序设计.其具体底层实现机制将在后文讨论.

2.2 一个简单的例子

如图 1 所示,首先为系统销售的图书建立基本的 Book 类,其中,每种图书记录了书名、作者、出版社、价格、库存数量等属性信息,系统会为这些属性自动生成相应读写方法.假设销售商现在要举行促销活动,对某一出版社的所有图书折价销售.为实现此需求,最简单直接的方法是修改该出版社所有图书的价格.但该方案的缺点在于,当促销活动结束时还需要将所有相关图书的价格重新设定,操作比较繁琐.另外,也可以采用 OOP 的继承机制为 Book 类定义一个子类,专门用于表示所有折价图书.但正如本文前面所讨论的,OOP 系统中的对象无法改变其所属的类,一旦促销活动结束,这些图书并不能直接转变为 Book 类的对象.

```
class Book {
   String title, author, publisher;
   int price, quantity;

   …
}

@Role(source="Book",inherit={".*"})
class DiscountedBook {
   int getPrice() {
      source.getPrice()*0.8;
   }
}
```

```
aBook=new Book("DR","TANG","WHU",20,5);
…

if (aBook.publisher="WHU")
      aDiscountedBook=new DiscountBook(aBook);

if (Time.now≥promotion_start_time &&
   Time.now≤promotion_end_time)
      println(aDiscountedBook.price);   //should be "16"
else
      println(aBook.price);             //should be "20"
…
```

Fig.1 Definition and usage of static role class

图 1 静态角色类的定义和使用

但是,从角色模型的角度考虑,可以认为相关图书在促销期间实际上担当了一种"促销图书"的角色,它们应该按其促销价销售,但其原始价格并不应该发生变化.基于这一考虑,我们可以为 Book 定义一个角色类 *DiscountedBook* 用于实现上述需求.关于 DR 中类的定义,有以下几点需要说明:

- 缺省情况下,所有类中的实例变量具有私有访问权限,而方法则具有公共访问权限.系统会自动为所有实例变量生成相应读写方法,并将对实例变量的访问转换为对读写方法的调用.
- 角色类应该在其类声明前通过标注*@Role* 表明此处定义的是一个角色类.
- 可以通过*@Role* 标注中的属性 *source* 声明该角色类的源类.
- 可以通过*@Role* 标注中的属性 *inherit* 声明该角色类从源类中所继承的方法(此处继承的含义与传统 OOP 中的继承含义有所不同,在此只是沿用了该术语,见后续解释).该属性的值是一个字符串数组,可以在其中显式列出源类中的方法名称,或用正则表达式来匹配源类中的多个方法名称. *DiscountedBook* 的定义中使用了一个正则表达式匹配源类中的所有方法(注意,由于所有的实例变量都是私有的,需要通过访问方法进行访问,因此此处不需要考虑对实例变量的继承问题).
- 可以在角色对象上直接调用从源类中所继承的方法,就好像它们是直接定义在角色类中一样.实际上,角色类中并不包含这些方法的定义,角色对象会将收到的消息转发给源对象进行响应.对于那些未被角色类所继承的方法,则表示角色类屏蔽了这些方法,在角色对象上对其调用则会抛出异常.
- 角色类可以定义自己的实例方法对源类的行为进行扩充,但如果角色类中定义了与源类同名的方法,则表示该方法被角色类所重写,即角色类可以改变源类中原有的行为.此时,如果需要在重写方法中调用源类的同名方法,则可以在方法名前增加 *source* 关键字前缀,如图 1 所示.其中,在 *DiscountedBook* 中重写了源类中对 *price* 属性的访问方法,该方法通过调用源类中的同名方法得到图书的原始价格,并对其进行相应折扣后,将新价格返回给调用者.

至此完成了一个简单角色类的定义.图 1 中的余下部分演示了可能的使用代码.首先创建所需的 Book 对象,

如果某 Book 对象的 *publisher* 属性符合指定的出版社名称,则为其创建一个 *DiscountBook* 类的角色对象,在促销活动期间,使用该角色对象获得图书价格,否则返回该图书的原始价格.这种使用方式称为角色对象的手工创建方式,通过显式编写代码完成角色对象的创建以及与源对象的绑定.其优点在于简单明了,实现机制也比较简单,是大多数角色模型都能提供的功能.另一方面,其缺点主要包括以下几点:

- 角色对象的创建需要显式编码,如果该出版社有很多图书,或同时为多个出版社的图书进行促销,则需要编写大量角色对象创建代码.促销活动结束后还需要编写大量清理代码来销毁角色对象;
- 在很多角色模型中,消息只能由角色对象单向转发给源对象进行响应,而不能由源对象自动转发给角色对象.为此,需要在外部程序调用中明确区分对源对象或角色对象的访问.如果当前时间在促销期内,则访问该图书的角色对象获取促销价格,否则访问源对象获取原始价格.在此情况下,需要根据角色对象的使用而改动外部程序的调用逻辑.而这种改动可能涉及到系统中的多个位置,并且在促销条件发生变化的情况下造成系统修改非常繁琐.

上述缺点也是静态角色模型的主要特点,它们限制了角色模型的应用.由于在系统业务逻辑中混杂大量对角色对象的控制和访问逻辑,使得角色对象的使用对用户不够透明,可能反而使得程序设计变得复杂.

2.3 使用动态角色类

针对第 2.2 节所讨论的问题,可以将 *DiscountedBook* 定义为动态角色类,这样就可以利用 DR 中的动态机制实现角色对象的动态创建和销毁过程,以及源对象与角色对象之间的消息双向转发.修改后的 *DiscountedBook* 类定义如图 2 所示(Book 类的定义同图 1).

```
@Role(source="Book",inherit={".*"},
      where="publisher=='WHU' &&
             Time.now≥Time.local(2010,3,1) &&
             Time.new≤Time.local(2010,3,31)")
class DiscountedBook {
  int getPrice() {
    source.getPrice()*0.8
  }
}
```

```
aBook=new Book("DR","TANG","WHU",20,5)
…
//suppose current date is 2010-3-15
…
println(aBook.price )      //should be "16"

aBook.publisher="WHUT"
println(aBook.price )      //should be "20"
…
```

Fig.2 Definition and usage of dynamic role class

图 2 动态角色类的定义和使用

在图 2 中,我们对角色类 *DiscountedBook* 的定义进行了部分修改,在*@Role* 标注中利用 *where* 属性增加了角色对象创建条件,该属性的值是一个逻辑表达式组成的字符串.角色对象创建条件意味着当系统中的某一对象 *s* 符合与其所属类 *S* 相关的角色类 *R* 中所定义的角色对象创建条件时,系统会为 *s* 创建一个属于该角色类的角色对象 *r*,并将其绑定到 *s* 上.此时,*s* 称为 *r* 的源对象.另一方面,如果 *s* 不再符合相关角色类中的创建条件,则系统会自动解除 *s* 与 *r* 之间的绑定,角色对象会被系统自动销毁.

针对图 2 中右栏所示程序代码,系统首先创建一个 Book 类的对象 *aBook*,并将其 *publisher* 和 *price* 属性分别设置为"WHU"和"20".假设在当前日期中该书发生了销售行为,系统需要向 *aBook* 发送 *price* 消息以获取其价格.由于此时 *aBook* 的 *publisher* 属性以及系统时间都符合 *DiscountedBook* 类中所定义的创建条件,因此 *aBook* 对象应该具有一个与其绑定的 *DiscountedBook* 类的角色对象.在此情况下,*aBook* 在响应对 *price* 属性的访问时,会发现在该角色对象中重写了 *price* 属性的访问方法.因此,*aBook* 不会自己响应对 *price* 属性的访问,而是将其转交给相应角色对象.而该角色对象中的 *getPrice* 方法则通过调用其源对象上的同名方法获得了 *aBook* 的原始价格,并经过折扣计算后返回给最初的调用者.此后,假设系统修改了 *aBook* 的 *publisher* 属性的值,并再次访问其 *price* 属性获取其价格信息.由于新的 *publisher* 属性值使得 *aBook* 不再符合相关创建条件,因此系统会自动销毁其在 *DiscountedBook* 类中的角色对象.此时,*aBook* 不再属于促销图书,对其 *price* 属性的访问会由其自己响应并返回原始价格.

需要强调的是,在上述使用动态角色类的例子中,角色对象的所有行为对于外部调用者而言都是透明的.外

部调用者并未主动创建任何角色对象,甚至不知道这些角色对象的存在,所有消息都是针对源对象发出的.基于DR 中的动态机制,角色对象会在适当的时候被自动创建或销毁,并且能够对源对象的行为进行扩充或修改.而对于外部调用者而言,在不必修改源类代码的情况下,源对象的行为就能根据某些状态条件的变化而动态更新或恢复,这些都是动态角色模型所应该具备的基本功能.对于图 2 中的例子而言,如果想改变图书的促销规则,只需要在 *DiscountedBook* 类的定义中修改相应创建条件,而不必再为此修改系统中的其他程序逻辑,这也使得上节中讨论的问题都能够得到解决.

图 1 和图 2 中的例子只是使用 DR 进行程序设计的最基本应用.使用 DR 同样能够解决更加复杂的编程需求,如角色链或角色树的问题.本文后面将结合动态角色模型的实现机制对此分别描述,我们将看到 DR 的实现机制为以上这些问题都提供了一种简洁统一的解决方案.

3 动态角色模型的实现机制

当前的 DR 实现版本基于 Groovy[3]语言开发,这种 OOP 语言的动态特性和强大的反射功能为 DR 的实现提供了很多便利.但需要强调的是,DR 的核心实现机制是独立于语言的,也就是说,即使采用其他 OOP 语言,如Java,C++等,也能完成 DR 的实现,只是在一些语法或实现细节上需要根据语言特性进行调整.这些将留待后续研究工作详细分析和设计,本节主要介绍 DR 的核心实现机制.

3.1 关于不同功能实现机制的讨论

在传统 OOP 中,消息由调用者发送给接收对象,并由接收对象响应后将执行结果返回给调用者.这种消息响应机制简略描述于图 3(a)中,其中,*anObj* 作为接收对象能够对其收到的消息 *anObj*.*m*1()和 *anObj*.*m*2()作出响应(注意,该图省略了接收对象根据继承体系向上搜索方法的过程).基于该消息响应机制所实现的角色模型基本结构如图 3(b)所示,其中,*aSource* 是一个源对象,*aRole* 是与其绑定的一个角色对象,其内部保存了对源对象的一个引用.但是,该图所示结构无法满足 DR 中对角色模型的动态性要求,这些问题主要集中在以下 3 个小节所讨论的功能特性上.

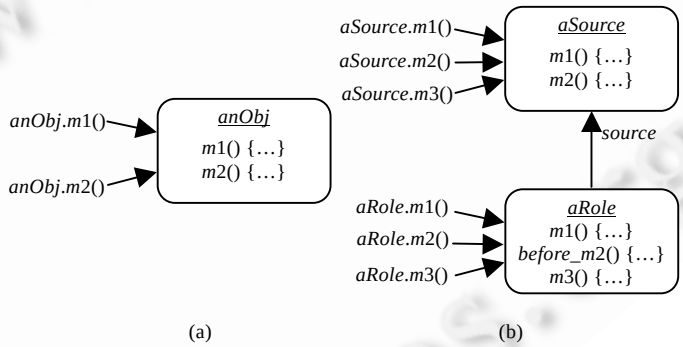


Fig.3 Role model based on traditional object-oriented model

图 3 基于传统面向对象机制的角色模型

3.1.1 角色对象向源对象的消息转发机制

角色对象向源对象的消息转发是角色模型的基本功能,如图 3(b)所示.对这一需求最直接的解决方案就是在角色对象中保留其对源对象的引用,这一引用可以在角色对象创建时一并生成.作为对源对象的扩展,角色对象不必具有所有的业务功能,而是将对核心业务功能的调用转发给其源对象进行处理.这一方式就是典型的代理转发(delegation)机制[6],它在大多数角色模型的实现中被采用,本文第 5.1 节中还将对此进行讨论.在 DR 的实现中也同样采用了这一机制,但在以下两点上需要注意:

- DR 中设置了回调(callback)机制,可以在消息向源对象转发之前或由源对象返回之后,通过角色对象

中的钩子(hook)函数分别执行一段额外的程序逻辑,从而更加灵活地扩展源对象的功能.在图 3(b)中,如果将消息 *aRole*.*m*2()发送给角色对象 *aRole*,由于其并未提供 *m*2()方法的实现,因此该消息将被转发给 *aRole* 的源对象 *aSource* 进行响应.在此转发过程中,因为钩子函数的存在,系统会在 *aSource* 执行 *m*2()方法之前首先执行 *aRole* 中的 *before_m*2()方法.这种回调机制甚至能够根据运行时状态动态替换掉对源对象的方法调用.

- DR 除了能够显式地创建角色对象以外,还提供了角色对象的动态创建机制,但如何引用这些自动生成的角色对象却是一个问题.如图 3(b)所示,假设 *aSource* 的角色对象是通过动态创建机制生成的,系统在运行前并不知道该角色对象的存在,因而也不可能事先声明一个名为 *aRole* 的变量并通过赋值代码来保存对该角色对象的引用,系统只会在内部记录该角色对象的 *oid* 及其与源对象之间的绑定关系.对于外部调用者而言,它们无法获得角色对象的直接引用,因而也无法向其直接发送消息,也就不可能完成对角色对象的功能调用.关于这一点及相关问题,在以下章节还将深入讨论.

3.1.2 源对象向角色对象的消息转发机制

当源类中的部分方法被角色类重写时,则意味着角色对象可以改变源对象的原有功能,因此对源对象的相应调用应由角色对象中的重写方法进行响应,这也是 DR 中源对象向角色对象转发消息的含义所在.而且这还能够帮助解决前面讨论的角色对象动态创建机制所带来的引用问题,因为对于角色对象上的功能调用就可以通过源对象来完成.在图 3(b)中,角色对象 *aRole* 重写了源对象 *aSource* 的方法 *m*1(),为方法 *m*2()增加了执行前逻辑,并新增了方法 *m*3().由于角色对象中的程序逻辑并不在源对象的继承链上,因此,如果消息不能由源对象转发给角色对象,这将导致对消息 *aSource*.*m*1()的响应仍是源对象中的方法 *m*1(),对消息 *aSource*.*m*2()的响应不会触发角色对象中定义的执行前逻辑,对消息 *aSource*.*m*3()的响应则可能会导致系统异常.对于这一问题,一种容易想到的解决方案是,在每个源对象中保留其对角色对象的引用,并将需要转发的消息通过这个引用转发给角色对象.但是,这种方案的可行性需要仔细讨论,详见以下分析:

- 若要在每个源对象中保留对角色对象的引用,则需要为源类增加新的属性.如果这些源类是用户的自定义类,则问题相对简单,可以通过修改源代码增加所需属性;但是如果这些源类是系统类或基础类,由于通常无法获得其源代码,要为它们增加新的属性并不容易.即使能够获得所需源代码,但对系统类或基础类进行改动也可能导致兼容性问题.在 DR 的实现语言 Groovy 中,虽然可以利用动态性方便地在 *Object* 类中增加新的属性,但基于上述考虑,我们也没有在 DR 的实现中采用该方案.
- 仅仅在源对象中保留对角色对象的引用还不够,实际的消息转发还需要由代码来完成.接下来的问题是,执行转发的代码应该保存在什么地方?一种可能的方案是修改源类的代码.当源类中的方法被角色类重写时,则由系统自动修改源类中相应方法的方法体,使其能够根据所保留的引用去调用角色对象中的新方法,从而实现消息转发.这种机制在很多语言中都能够实现,例如,在 Java 平台中利用字节码修改工具就能在不改变源代码的情况下直接修改方法体字节码.但这种方案的缺点在于:1) 不利于实现反向过程.如果角色对象解除了与源对象的绑定,则源对象应恢复其原有功能,但之前对方法体的修改可能导致恢复过程无法实现.虽然在 Java 平台上可以利用实时编译技术基于初始源代码动态重新生成原有的方法体字节码,但这需要依赖于特定平台,并且源代码也不总是能够轻易获得的;2) 直接修改源类中的方法体会影响到该类的所有对象,但该类的所有对象并不一定都具有与其绑定角色对象,该方案无法解决这一差异性问题.虽然在一些动态 OOP 语言中还可以利用某些特定技术(例如,为类中的方法赋予别名等机制)来解决上述问题,但由于都过分依赖每种语言自身的特性而难以在其他语言平台上实现,故而没有被 DR 的实现机制所采用.

3.1.3 角色对象的动态创建机制

在角色对象动态创建机制的实现中,如何监测角色对象创建条件是否满足,以及具体由谁负责创建相应的角色对象则是需要考虑的问题.首先想到的解决方案可能是线程机制,可以为每个角色类创建一个线程去持续检测相应源类中所有对象的状态,一旦其满足创建条件就为其创建相应角色对象.但这一方案显然不具有可行

性.且不论线程间通信对系统复杂性所造成的影响,单就每个线程的检测频度和监控范围就难以确定.检测间隔或监控范围过小则会增加系统开销,而检测间隔或监控范围过大则可能无法及时创建角色对象.

另一种可能的方案则是修改源类的定义,在每个方法的末尾增加检测逻辑.由于方法调用可能改变对象状态,因而在每个方法的末尾检测当前对象的状态是否符合创建条件并作出响应,则可以避免前述线程机制可能发生的创建延迟问题.但这种方案的缺点在于:1) 需要改动源类中所有的方法体,并且当角色类的定义改变或撤销时还必须同步更新所有的方法体,操作比较繁琐;2) 需要保证所有对源对象属性的访问都通过访问方法完成,如果用户在外部直接修改了源对象中公共属性的取值,则对象状态的改动可能无法被监测到.

3.2 基于前置对象的实现机制

针对第 3.1 节讨论的问题,虽然提出了若干解决方案,但都无法根本解决,并且解决方案的多样性使得整个系统的实现变得更加复杂.本文提出了一种基于前置对象 prepositive object 的实现机制.它能够以简洁统一的方式解决上述所有问题,并能够方便地实现复杂角色体系.本节介绍该机制的核心思想.

3.2.1 为对象设置前置对象

首先,在 DR 的实现中对传统 OOP 模型进行了一点改造.我们利用了类似 proxy pattern[8]的设计模式,所有对象(除前置对象自身以外)在系统中都有一个前置对象,而原有对象被称为其前置对象的核对象(core object),如图 4 所示.其中,圆形表示一个前置对象,其特性主要包含以下几点:

- DR 中一个完整的对象结构由前置对象及其核对象组成,它们各自都是传统 OOP 中的普通对象.当调用某个类的构造方法时,系统会首先创建相应核对象,并接着为其创建一个前置对象,并在前置对象中设置对核对象的两个引用,即 *core* 和 *next*.其中,*core* 指针在对象的整个生命周期内都不再更改,即总是指向该前置对象的核对象.
- DR 中,对象间的引用都指向其前置对象.也就是说,指向对象的变量实际上引用的是该对象的前置对象,而每个前置对象的核对象对于外部来说是不可见的(注意,此处“对象间的引用”指的是一般对象之间的引用,角色对象对源对象的引用机制与此不同,将在第 3.2.2 节详细介绍).这意味着调用者发送的消息并不会直接传递给核对象,而是首先由其前置对象所接收.
- DR 中的所有前置对象都具有相同功能,它们收到任何消息之后会接着将消息转发给其 *next* 指针所指向的后续对象.在初始情况下,该后续对象就是其核对象.后续对象对收到的消息进行响应后,将结果返回给前置对象.但是,前置对象在转发消息之前会在系统中检查所有与其核对象所属类相关的角色对象创建条件,若有满足的条件,则首先创建相应角色对象,然后再进行消息转发.
- 如果前置对象收到的消息名称前具有 *core* 前缀,则前置对象不再沿 *next* 指针转发消息,而是将该消息转发给其 *core* 指针所指向的核对象,并且不会在转发前进行角色对象创建条件的检查.

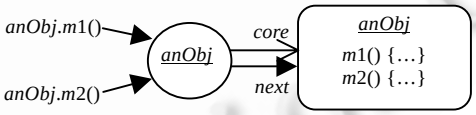


Fig.4 Create prepositive object for object

图 4 为对象设置前置对象

通过上述介绍不难理解,前置对象的功能和结构都相对简单.在具体实现中,可以利用 OOP 系统的 *method_missing* 机制[1]实现前置对象所需的消息转发功能.所有前置对象的功能逻辑都是相同的,因此只需要定义一个统一的前置对象类,而不必再为每个类定义专门的前置对象类,这也使得系统实现保持精简.虽然需要修改 OOP 系统原来的对象创建过程以便为每个对象同时创建一个前置对象,但这种修改在很多 OOP 语言中也并不复杂(例如,在 Java 中可以通过 *class-loader* 机制完成该需求).需要强调的是,使用前置对象的 DR 系统仍然可以兼容传统 OOP 系统的所有功能.由于前置对象仅仅是执行了消息转发,每个核对象在收到转发来的消息时,

首先仍然按照 OOP 系统原有的消息响应机制在其继承体系中进行方法匹配,如果找不到所需方法再按照 DR 的相应机制进行处理.这样,在还没有创建任何角色对象的情况下,整个系统的程序执行过程和结果与传统 OOP 系统是完全一致的,这也使得 OOP 的设计方法在 DR 中仍然可以使用.

至于前置对象对系统效率的影响则主要体现在两个方面:一是系统空间的开销.前置对象不可避免地会消耗内存空间,但鉴于前置对象的功能和结构相对简单,其空间消耗量也很小,并且在当前硬件条件下,这些空间消耗应该是可以接受的;二是系统效率的降低.使用前置对象为消息响应过程增加了两个环节——转发消息前对角色对象创建条件的检索过程以及向后续对象转发消息的过程,相对来说,后一环节由于是指针操作不会过多影响系统效率,而前一环节则需要在实现上进行细致设计,尽量减少每次检索时所需匹配的条件数量.这一问题的解决也存在着若干不同的优化策略,但由于篇幅原因,本文不对此进行详述,我们将在后续研究工作中对此进行详细分析和改进.一种可以采用的方案是,可以把 DR 作为对传统 OOP 系统的补充,或者是支持敏捷开发过程的一种工具.在软件系统主要部分用成熟 OOP 技术进行开发的情况下,对于一些特殊的动态功能或需要快速实现的原型系统可以利用动态角色技术进行实现.这样可以减少角色类的使用数量,也不必为系统中的所有对象创建前置对象,而只为那些可能需要绑定角色对象的源对象创建前置对象.这样会使得前置对象的数量显著减少,而前置对象对系统效率的影响相对于其所提供的动态能力也是能够接受的.

3.2.2 利用前置对象实现动态角色机制

本节将介绍如何利用前置对象机制实现角色模型所需的动态功能.在图 5(a)中,*aSource* 作为前置对象,其 *next* 与 *core* 指针在初始状态下都指向其核对象,即 *ASource* 类的实例 *aSource_core*.该前置对象会在消息转发之前检查所有与 *ASource* 类相关的角色对象创建条件.假设在此过程中 *aSource* 检测发现相关角色类 *ARole* 中的创建条件被满足,则系统会按照对象创建过程首先创建 *ARole* 类的实例 *aRole_core*,紧接着为其创建一个前置对象 *aRole*,并将其 *core* 和 *next* 指针都指向 *aRole_core*,如图 5(b)所示.角色对象 *aRole_core* 创建完毕后,系统还需设置该角色对象的 *source* 指针,使其指向负责创建该角色对象的那个前置对象(即 *aSource*)的 *next* 指针的相同目标(即 *aSource_core*).之后,系统还要更改 *aSource* 的 *next* 指针,使其指向新创建角色对象的前置对象(即 *aRole*).至此,角色对象与源对象的绑定过程结束,如图 5(c)所示.

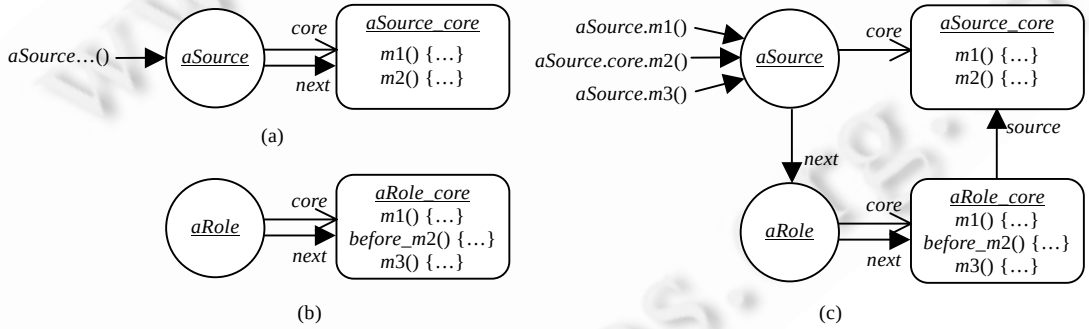


Fig.5 Dynamic role model based on prepositive objects

图 5 利用前置对象实现动态角色机制

在图 5(c)中,当前置对象 *aSource* 转发其收到的 *aSource.m*1()消息时,该消息将沿 *next* 指针转发给 *aRole*,并接着由 *aRole* 再沿 *next* 指针转发给 *aRole_core*.由于 *aRole_core* 中具有方法 *m*1()的实现,因此该消息会由 *aRole_core* 响应后,再将结果沿原路返回给最初调用者.这种执行结果与之前所期望的结果是一致的,由于 *aRole_core* 重写了 *aSource_core* 中的同名方法,因此角色对象中的程序逻辑应代替源对象中的程序逻辑,这样也就实现了由源对象向角色对象的消息转发机制.同样,对于前置对象 *aSource* 收到的消息 *aSource.m*3(),系统不会因为 *aSource_core* 中没有对方法 *m*3()的定义而报错,该消息将被 *aRole_core* 中的方法 *m*3()所响应,调用者不会因为没有对角色对象的直接引用而无法调用其中的消息.对于发送给前置对象 *aSource* 的消息 *aSource.m*2(),经

过 *aRole* 的转发到达角色对象 *aRole_core* 后,由于其中并不包含 *m*2()方法的实现,因此系统将继续沿角色对象的 *source* 指针将该消息转发给 *aRole_core* 的源对象.在此转发过程中,会因为回调机制和钩子函数的存在,而在 *aSource_core* 执行 *m*2()方法之前首先执行 *aRole_core* 中的 *before_m*2()方法.

另一方面,如果希望直接调用源对象 *aSource_core* 中的方法 *m*2(),而不希望在传递过程中触发 *aRole_core* 中的 *before_m*2()方法,则只需修改消息调用格式,将 *aSource.m*2()改为 *aSource.core.m*2()就可以了.这意味前置对象会将消息直接转发给 *core* 指针指向的核对象,这也是为什么我们在前置对象中保留两个指针的原因.

基于前置对象机制,角色对象的动态销毁也变得容易实现.一旦前置对象在转发消息之前发现某个已存在的角色对象的创建条件不再被满足,前置对象只需调整其 *next* 指针,使其指向该角色对象的 *source* 指针所指向的对象就可以了.在图 5(c)中,一旦 *aRole_core* 的创建条件不再满足,则 *aSource* 的 *next* 指针将重新指向 *aSource_core*,该角色对象会因为不再被引用而被垃圾回收机制自动销毁.

通过以上对角色对象的创建、绑定及销毁过程的描述可以发现,前置对象机制为所有过程提供了极大的方便,用户无须改变任何源类的定义,也无须从根本上改变 OOP 系统的对象结构和消息响应机制就可以实现 DR 所需的动态功能.同时,动态创建和动态销毁过程的结合使用也使得源对象的角色动态迁移机制成为可能.除此之外,在第 4 节我们还将看到前置对象能够为更复杂的角色体系提供支持.

4 利用前置对象实现复杂角色体系

图 5 所示的例子只是角色模型应用中最简单的情况,即一个源对象只有一个角色对象.但在实际应用中,真实情况会更加复杂.例如,用户可以为角色类再定义角色类,使得多个角色对象形成链状结构;也可以在一个源类上定义多个角色类,使得一个源对象同时有多个角色对象;或者定义有多个源类的角色类,使得一个角色对象可能同时有多个源对象.而前置对象机制能够为这些需求提供简洁、统一的实现.

4.1 角色链的实现

考虑图 2 中关于图书促销的例子,如果销售商希望在促销活动中每销售一本特定作者的图书,都向该作者发送一份电子邮件以告知其图书销售情况.针对该需求,可以再为 *DiscountedBook* 类定义一个角色类,在其中通过修改结账操作使得每销售一本指定图书时都能完成上述功能.鉴于篇幅原因,此处不列出该实例的具体代码,但是不难理解,该实例的类层次形成了一种角色链,如图 6 所示.也就是说,如果为角色类 *ARole* 定义了子角色类 *ASubRole* 及相关创建条件,则当 *aRole_core* 的状态满足该条件时,应该为其创建相应的子角色对象并完成绑定,从而在源对象上形成角色链结构.

通过图 6 不难理解,DR 中角色链的实现不需要任何专门机制,整个过程与图 5 中创建单一角色对象的过程完全一致.*aRole_core* 的前置对象 *aRole* 在转发任何消息之前,同样会检查所有与 *ARole* 类相关的角色对象创建条件.一旦有条件被满足,*aRole* 就会按照本文第 3.2.2 节中描述的流程创建新角色对象 *aSubRole_core* 及其前置对象 *aSubRole*,并将 *aRole_core* 作为新角色对象的源对象进行绑定,从而完成整个角色链的动态创建.从理论上讲,这种动态创建的角色链可以具有任意深度,而且系统会自动维护角色链的组成.一旦链条上的某个角色对象的创建条件不再被满足,系统会根据动态销毁过程把该角色对象及其所有子角色对象都从角色链上自动断开.这一处理机制是合理的,因为如果一个角色不再存在,那么所有依赖该角色的角色也不应该再存在.

基于前置对象的角色链也不会对消息响应造成混乱.例如在图 6 中,发送给 *aSource* 的消息 *aSource.m*1()经过 *aRole*,*aSubRole* 后到达 *aSubRole_core* 并由其进行响应.这意味着如果角色链上的多个角色对象都重写了上层源对象中的同名方法时,则最新的那个方法将被执行,而之前的方法会被新方法所覆盖.对于源对象中那些没有被其任何角色对象所重写的方法,消息调用在随着 *next* 指针到达角色链的末端后,会再沿 *source* 指针传递到该源对象处,这也保证了消息响应机制能够正常运行.

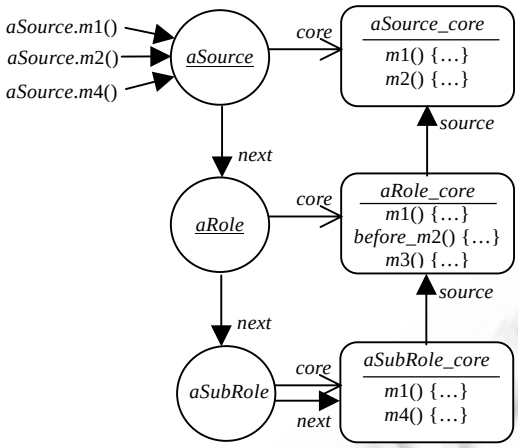


Fig.6 List of role objects

图 6 源对象上的角色链

4.2 角色树的实现

对第 4.1 节提到的关于图书促销的例子进行一点改动,如果销售商希望无论是促销活动还是正常销售行为,只要每销售一本特定作者的图书,都向该作者发送一份电子邮件.针对该需求,可以将 Book 类作为源类,并为其再定义一个角色类,在该角色类中通过修改结账操作使得每销售一本指定的图书时都能够执行上述功能.鉴于篇幅原因,此处不列出该实例的具体代码,但是不难理解,与图 6 中所示情况不同,Book 类此时具有两个直接相关的角色类,而对于那些同时满足这两个角色类创建条件的图书,则会具有两个直接绑定的角色对象,即可能基于一个源对象形成角色树结构,类似于图 7 所示情况.在此情况下,如何创建角色树以及传递消息则需要仔细考虑.

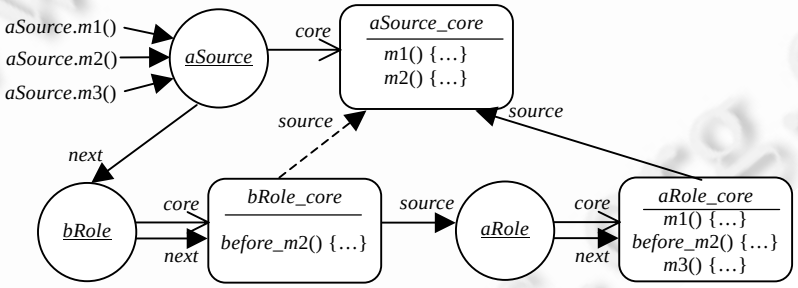


Fig.7 Tree of role objects

图 7 源对象上的角色树

前置对象机制同样可以解决这一问题.如图 7 所示,系统在图 5(c)的基础上为 *aSource_core* 创建了新的角色对象 *bRole_core*,但如何将其与源对象绑定起来?首先想到的结果可能是:在 *bRole_core* 创建完毕后,将其 *source* 指针指向 *aSource_core*(图 7 中虚线所示).但这种方案的问题在于,*aSource_core* 还有其他直接角色对象 *aRole_core*.如果将 *bRole_core* 的 *source* 指针直接指向 *aSource_core*,那么发送给 *aSource* 的所有消息在沿转发路径到达 *aSource_core* 之后,由于已经到达顶层源对象,因此不再会被转发给前置对象 *aRole* 及其之后的所有对象.也就是说,*aRole_core* 不可能再收到任何由 *aSource* 转发的消息.虽然 *aSource_core* 的角色对象 *aRole_core* 中包含方法 *m*3(),但发送给 *aSource* 的消息 *aSource.m*3()根本无法到达 *aRole_core*,这不符合所期望的结果,我们希望绑定在同一个源对象上的多个角色对象都能对源对象的功能进行扩展.

回顾第 3.2.2 节描述的角色对象创建完成后与源对象的绑定过程,它分为两个步骤:1) 首先将角色对象的 *source* 指针设置为与创建该角色对象的那个前置对象的 *next* 指针相同的值;2) 再将上述前置对象的 *next* 指针指向新角色对象的前置对象.根据这一过程,对照图 7 中的情况,在绑定新角色对象之前,负责创建 *bRole* 及 *bRole_core* 的前置对象 *aSource* 的 *next* 指针是指向 *aRole* 的,按照步骤 1)的要求,应该将 *bRole_core* 的 *source* 指针指向 *aRole* 而不是 *aSource_core*,然后再按照步骤 2)的要求将 *aSource* 的 *next* 指针转而指向 *bRole*.最终结果如图 7 中的实线所示.在这种情况下,发送给 *aSource* 的消息 *aSource.m*3()将沿转发路径经过 *bRole*,*bRole_core*, *aRole* 后到达 *aRole_core* 并由其进行响应.而发送给 *aSource* 的消息 *aSource.m*2()将沿转发路径经过 *bRole*, *bRole_core*,*aRole*,*aRole_core* 后最终达到 *aSource_core* 并由其进行响应.转发过程中,在 *bRole_core* 向 *aRole* 转发以及 *aRole_core* 向 *aSource_core* 转发的时候会依次触发 *bRole_core.before_m*2()和 *aRole_core.before_m*2()方法,即多个角色对象按照绑定次序共同扩展了同一源对象中的特定功能.另外,根据角色树的生成及绑定过程,如果多个角色对象都重写了源对象中的同名方法,则后绑定的角色对象将覆盖先绑定的角色对象中的相应方法,这与第 4.1 节中关于角色链的消息处理原则也是相同的.

需要强调的是,如图 7 所示,在 DR 的当前版本中,当一个源对象有多个直接角色对象时,我们将这些角色对象看作是对源对象的共同扩展,它们之间是一种按照创建次序而前后链接的兄弟关系,并共同影响对源对象的消息调用,这对于本节的例子是适当的.但是在其他应用中,可能希望一个源对象的多个直接角色对象之间是相互独立的,它们在不同的应用上下文中独自扩展或改变源对象的功能,从而提供更加灵活的动态角色模型功能.关于这一点,我们正在 DR 的后续版本中加以完善,本文不再对此进行详细分析.

4.3 多源角色对象的实现

在 DR 中还可以提供对多源角色对象的支持,一个角色对象可以同时具有多个源对象,这能够为软件开发中一些具有聚合或者分组语义的应用提供便利.对这类角色对象的功能本文不作详细描述,但同样可以利用前置对象实现其消息转发和响应机制.如图 8 所示,其中描述了对聚合角色对象的消息处理情况.当为 *aSource_core* 和 *bSource_core* 生成聚合角色对象后,所有发送给两个源对象的消息都会被其前置对象转发给 *aJoinRole*,并进而由角色对象 *aJoinRole_core* 进行处理.该角色对象可能会自己响应消息,或根据内部机制选择某一源对象进行响应.在更复杂的需求下,还可以为聚合角色对象创建其自己的角色对象,或将多个聚合角色对象再进行聚合而生成新的角色对象.而所有的过程都可以在前置对象机制的控制下,保证消息在角色树中的正确转发.

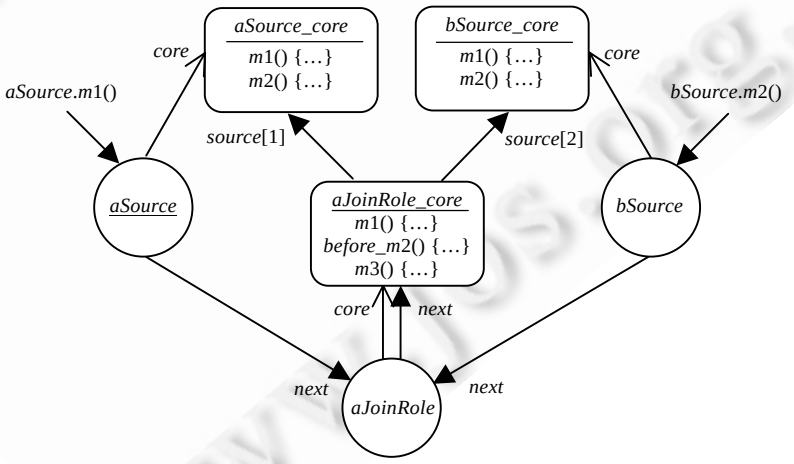


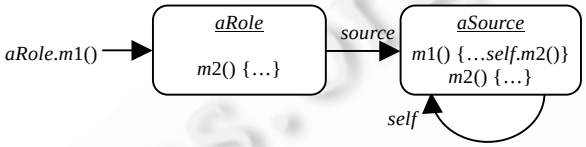
Fig.8 Using prepositive objects for joined role objects

图 8 聚合角色对象的对象间关联

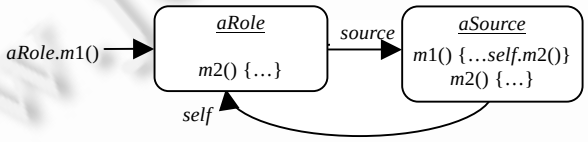
5 相关问题及后续工作

5.1 对self指针的处理

Herrmann 在 ObjectTeams[11]的实现中曾讨论了对 *self* 指针(有些语言中也称为 *this* 指针)的处理问题,这也是所有 OOP 系统中的一个重要概念,但在基于类(class-based)或基于原型(prototype-based)的 OOP 系统所开发的角色模型中则存在不同处理方式.如图 9 所示,其中,*aRole* 为一个角色对象,*aSource* 是 *aRole* 的源对象.按照一般角色模型的功能需求,*aRole* 会将其所收到的且不能被其自己响应的消息都转发给源对象 *aSource*.此时,如果 *aSource* 的某个方法体中需要访问 *self* 指针,则有可能存在不同的引用目标.图 9(a)是在基于类的 OOP 系统中的处理情况,*aSource* 中的 *self* 指针指向处理当前消息的那个对象自身,也就是 *aSource*.而图 9(b)是在基于原型的 OOP 系统中的处理情况,*aSources* 中的 *self* 指针指向最初接受消息的那个对象.由于 *aRole* 和 *aSource* 通常被视为逻辑上的一个整体,外界发送给它们的消息不管最后由哪个对象响应,都被视为是发送给这个整体的,因此将 *self* 指向最初接受消息的那个对象是合理的.为严格区分这两种对 *self* 指针的处理方式,通常将前者称为转发(forwarding),而将后者称为代理(delegation)[6].



(a) Forwording
(a) 转发



(b) Delegation
(b) 代理

Fig.9 Various mechanisms for self pointer
图 9 对 self 指针的不同处理机制

转发和代理这两种机制常常被混淆,但各自所表示的语义及其依赖的底层 OOP 系统是完全不同的,因此很难在同一个系统中共存.由于需要维护 *self* 的动态引用,所以在基于类的 OOP 系统中实现严格的代理机制也比较麻烦.为降低实现难度,当前绝大多数角色模型都采用图 9(a)所示的转发机制来实现角色对象向源对象的消息转发.但是在 DR 中实现严格的代理机制则非常简单,不必对底层系统作太多改动,也不必过多考虑 *self* 指针的动态维护问题.对于系统中具有前置对象的对象,只需在创建时将其 *self* 指针指向其前置对象就可以提供严格的代理机制(对于那些没有前置对象的对象,其 *self* 指针仍然指向其自身),如图 10 所示.

在图 10(a)中,发送给 *aSource* 的消息 *aSource*.*m*1()经过转发路径 *aRole*,*aRole_core* 之后会被 *aSource_core* 中的 *m*1()方法所响应,在 *m*1()的方法体中通过 *self* 指针调用 *m*2()方法.由于源对象 *aSource_core* 的 *self* 指针指向其前置对象 *aSource*,因此对消息 *self*.*m*2()的处理会由 *aSource* 再次转交给 *aRole*,并最终由 *aRole_core* 中的 *m*2()方法所响应.这样的结果符合 DR 的规则,角色对象中的重写方法覆盖了源对象中的同名方法,并且也实现了严格代理机制所期望的行为.另一方面,如果 *aSource_core* 的角色对象不再存在,则 *m*1()方法中的 *self*.*m*2()消息应由其自身的 *m*2()方法所响应.这一需求在 *aSource_core* 的 *self* 指针指向其前置对象的情况下仍然可以得到满足,如图 10(b)所示.由此可见,DR 中采用的前置对象机制为在基于类的 OOP 系统的角色模型中实现严格代理机制也提供了很大方便.

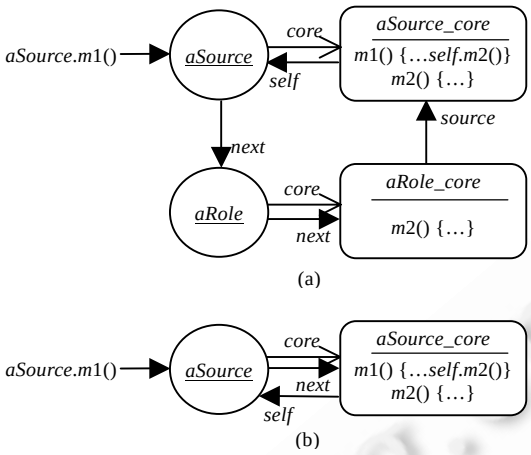


Fig.10 Using prepositive objects for delegation mechanism

图 10 利用前置对象实现 delegation 机制

5.2 对角色对象创建机制的改进

DR 的当前版本中有两种角色类:一种是具有创建条件的动态角色类,系统根据其创建条件自动创建或销毁相关角色对象;另一种是不带创建条件的静态角色类,其相关角色对象的创建和销毁都需要通过显式编写代码完成,即角色对象的手工创建机制.前置对象机制能够很好地在复杂角色体系中完成对这两类角色对象的链接及消息转发,但在角色对象的创建和销毁上存在一定的限制,见如下分析:

- 不能手工创建动态角色对象.由于动态角色类的定义中包含创建条件,而在手工创建动态角色对象时其创建条件可能并不满足,系统会阻止该创建过程并报错.即使系统允许强制创建并将其绑定到源对象上,也有可能因为创建条件不满足而在后续操作中立即被系统自动销毁.
- 静态角色对象必须手工销毁.由于静态角色类不包含创建条件,因此也无法利用自动检测机制销毁静态角色对象.如果用户也忘记销毁不应存在的静态角色对象,则有可能导致系统行为出现错误.

为解决上述问题,我们计划在 DR 的后续版本中对角色对象创建机制进行改进,其主要思想是,在角色类中增加对销毁条件的定义.角色类中可以同时包含创建条件和销毁条件,也可以只有任意一个.前置对象会分别检查与其核对象相关的创建条件和销毁条件,创建条件满足则生成角色对象,销毁条件满足则断开对角色对象的链接.如此一来,既可以手工创建那些包含创建条件的动态角色对象,只要其销毁条件不满足就可以存在于系统中;也可以自动销毁那些不带创建条件的静态角色对象,只要其销毁条件被满足就能被系统自动销毁.实际上,在分别支持创建条件和销毁条件的情况下,静态角色类和动态角色类之间也就不再存在明显分别,我们也能够以更加统一的机制控制所有角色对象.

5.3 其他后续工作

在 DR 的后续版本开发中,我们希望能够探索更多语言平台上的实现方式.从本文讨论的前置对象机制来看,该机制是独立于语言特性的,应该能够应用于所有的 OOP 系统.但在一些静态强类型 OOP 语言中,其动态性和开放性毕竟不如动态 OOP 语言灵活,因而会有更多的实现细节需要考虑.

另外,前置对象中的角色对象创建条件的动态检测机制也需要进一步的分析和优化(包括对销毁条件的支持机制).在角色类数量较多、创建条件较为复杂的情况下,如何使创建条件的存储和检索能够保持较高的效率?我们希望能够在后续工作中通过更多的理论和实验分析找到更好的方案.

随着近几年相关研究的进展,角色模型研究领域呈现出与面向方面程序设计(AOP)[13]研究领域相互融合的趋势.利用角色模型解决 AOP 中的问题,或利用 AOP 底层架构实现角色模型所需功能的相关研究都有所尝

试.本文所实现的动态角色模型也对此提供了基本的支持,在角色对象向源对象转发消息时可以利用回调机制和钩子函数插入附加程序逻辑,这与AOP中的通知(advice)机制非常相似,但当前DR中实现的功能还相对简单.从另一方面看,本文所描述的关于图书促销的例子也可以利用AOP的思想进行解决,这些都为DR用于AOP领域提供了契机.

6 结束语

角色模型的应用可以改善OOP模型由于数据封装所导致的对象灵活性问题.但基于OOP模型所设计和实现的角色模型却又可能受到OOP模型的限制而导致其动态性不足或实现机制过于复杂.本文提出了一种基于前置对象机制的动态角色模型,不但能够在传统角色模型的基础上提供更高的动态特性,并且为复杂角色体系及消息传递机制的实现提供了简洁、统一的方案.本文对实现过程中的相关问题也进行了详细分析和讨论.

需要强调的是,本文所描述的动态角色模型只是在OOP系统的对象创建机制上进行了有限改动,但仍然能够与传统OOP模型完全兼容,这主要得益于以下3点:

- 无须扩展底层OOP语言的语法结构或新增关键字来表示角色类和角色对象等概念;
- 并未改变OOP系统的个体对象结构,前置对象和核对象本身都是OOP中的标准对象;
- 并未根本改变OOP系统的消息响应机制,只是OOP系统不能响应的情况再由角色模型进行处理.

以上特性使得本文所描述的动态角色模型能够方便地集成到现有OOP系统中,这使得开发人员不必浪费在原有OOP系统中所积累的代码、技术和经验,这些都显著提高了角色模型的可应用性.

致谢 在此,我们向对本文的工作给予支持和建议的同行,尤其是武汉大学计算机学院彭智勇教授领导的讨论组的老师和同学表示感谢.

References:

[1] Thomas D, Flower C, Hunt A. Programming Ruby. 3rd ed., Dallas: The Pragmatic Programmers LLC, 2009.

[2] Lutz M. Learning Python. 4th ed., Cambridge: O'Reilly Press, 2009.

[3] Subramaniam V. Programming Groovy. Dallas: The Pragmatic Programmers LLC, 2008.

[4] Flatt M, Krishnamurthi S, Felleisen M. Classes and mixins. In: MacQueen DB, Cardelli L, eds. Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'98). New York: ACM Press, 1998. 171−183.

[5] Ungar D, Smith RB. Self: The power of simplicity. In: Meyrowitz N, ed. Proc. of the Object-Oriented Programming Systems, Languages and Applications 1987 (OOPSLA'87). New York: ACM Press, 1987. 227−242.

[6] Stein LA. Delegation is inheritance. In: Meyrowitz N, ed. Proc. of the Object-Oriented Programming Systems, Languages and Applications 1987 (OOPSLA'87). New York: ACM Press, 1987. 138−146.

[7] Fowler M. Dealing with roles. 1997. http://www.martinfowler.com/apsupp/roles.pdf

[8] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[9] Baldoni M, Boella G, Van der Torre L. Interaction between objects in PowerJava. Journal of Object Technology, 2007,6(2):5−30.

[10] Tamai T, Ubayashi N, Ichiyama R. An adaptive object object model with dynamic role binding. In: Roman GC, Griswold WG, Nuseibeh B, eds. Proc. of the 27th Int'l Conf. on Software Engineering (ICSE 2005). New York: ACM Press, 2005. 166−175.

[11] Herrmann S. A precise model for contextual roles: The programming language ObjectTeams/Java. Applied Ontology, 2007,2(2): 181−207.

[12] Peng Z, Li Q, Feng L, Li X, Liu J. Using object deputy model to prepare data for data warehousing. IEEE Trans. on Knowledge and Data Engineering, 2005,17(9):1274−1288. [doi: 10.1109/TKDE.2005.154]

[13] Tang Z, Peng Z. Survy of aspect-oriented programming language. Journal of Frontiers of Computer Science and Technology, 2010, 4(1):1−19 (in Chinese with English abstract). [doi: 10.3778/j.issn.1673-9418.2010.01.001]

附中文参考文献:

[13] 唐祖锴,彭智勇.面向方面程序设计语言研究综述.计算机科学与探索,2010,4(1):1−19. [doi: 10.3778/j.issn.1673-9418.2010.01.001]

唐祖锴(1977－),男,湖北武汉人,博士生,讲师,主要研究领域为软件工程,面向对象程序设计语言.

任毅(1973－),男,博士生,讲师,主要研究领域为软件工程,可信数据管理.

彭智勇(1963－),男,博士,教授,博士生导师,主要研究领域为面向对象程序设计语言,数据库.

崔晓军(1972－),男,博士生,副教授,主要研究领域为软件工程,Web 数据集成.