计算机科学1998Vol. 25№. 6

并发面向对象语言的继承异常问题研究*)

Some Researches on Inheritance Anomaly of Concurrent OO Languages

张 鸣 杨大军 吕 建

(南京大学计算机软件新技术国家重点实验室 计算机软件研究所 南京210093)

摘 要 In this paper, based on the difference between sequency and concurrency, the conflicts between concurrency and inheritance in concurrent object-oriented languages are analyzed. Then the phenomena, reasons and typical solutions of inheritance anomaly are surveyed. Finally, we propose a new method to solve the inheritance anomaly.

关键词 Object-orientation, Inheritance, Concurrent Object-orientation, Inheritance Anomaly

1. 引言

随着面向对象技术的发展,面向对象程序设计语言在软件设计、模块化、可扩充性、可复用性等方面给软件开发人员带来了很大的方便[1]。继承是顺序面向对象语言的一个基本特点,继承机制是面向对象语言的重要机制之一,是实现软件复用和可扩充的有效语言机制。Bertrand Meyer 曾指出纯 OO 语言的七个特性[2]:①模块化结构;②数据抽象(对象是抽象数据类型的实现);③自动内存管理(无用对象的回收不需程序员的干预);④每个非简单类型是类;⑤继承;⑥多态性和动态定联(程序实体可引用不止一个类的对象,不同类中的相同操作可以有不同的具体实现);⑦多继承和重复继承。

其中两次提到继承问题,由此可见继承是面向对象语言的一个重要特征。

由于应用对并行处理的需求,许多学者希望能将并发性与面向对象技术有机地结合起来。通常,在面向对象环境中引入对并发支持可以有两种途径:一种就是在现有面向对象环境的基础上再另外引入与并发性有关的一些概念,如进程、管程等,Smalltalk[3]中采用的就是这种方法,但由于对象与进程概念的不同,造成在问题建模和进行对象保护时会产生问题。另一种方法就是将对象与进程两个概念有机地统一成并发对象的概念,并将对象间消息传递与进程间交互也有机地结合在一起,Act-1和 ABCL 中采用的就是这种方法。无论采用何种途径,面向对象技术与并行机制结合的一个重要问题就是继承问题,即如何在并行面向对象语言中提供合适的继承机制。

很多支持并发的 OO 语言都是在原来语言的基础上融入并发的特性,但由于并发面向对象环境中可以同时有多个消息到达并且具有不确定性,致使并发和继承会产生冲突。为避免这种冲突,有的语言中就不提供对继承性的支持,比如 POOL-T[4]、COOL、Presto、Emerald 等[2],严格来说,这类语言不能算是面向对象的语言,而只能是基于对象的语言。另一途径就是象 Matroshka 中采用的途径,它仅支持静态继承,提出一种同步机制来避免这种冲突。虽然 POOL-T 以后的版本(POOL/R)中提供了有限制性的继承,但许多工作需由程序员来完成。

很显然,上述途径并不能真正地解决并发语言中的继承异常问题,因此,国际上对并发面向对象的继承异常问题进行了多方面的研究,提出了各种各样的解决方法。本文的主要工作就是从面向对象语言在顺序与并发环境中的区别出发,说明什么是继承异常、继承异常的现象、原因及解决方法,最后提出一种处理继承异常的新方法。

*) 本项研究受到国家杰出青年科学基金和攀登计划项目的资助。张鸣 硕士生,研究方向为面向对象语言、形式化方法。杨大军 博士生,研究方向为面向对象语言、形式化方法。吕建 教授,博士导师,研究方向为软件自动化、并行程序形式化方法、面向对象语言和环境等。

2. 顺序与并发

在顺序的面向对象环境中，对象间的消息传递具有单一性和确定性，因而消息的接收不需进行同步控制，即当对象 O 接收到消息 m 时，便产生对方法 m 的调用，相当于在顺序的对象体中有这么一个统一的"消息接收控制体"：

```
class class_name{
    //数据定义部分
    ……
    //方法定义部分
    m1( ){    ;
    ……
    mn( ){    ;
    //统一的消息接收控制体
    switch coming_message
    {
    case m1:call m1;
    ……
    case mn:call mn;
    }}
```

图1　顺序的"消息接收控制体"

在顺序环境中，由于该控制部分是统一的，所以进行类的定义时可以省略掉这一部分，子类中即使有新方法引入也不会影响该消息接收控制体（只是加几条 case 语句，对父类中继承过来的方法不会有任何影响）。

在并发环境中，由于可以同时有多个消息访问一个对象，并且对象之间的消息传递具有不确定性，所以为了保证对象的完整性和对象中数据的一致性，必须在对象中引入同步控制部分。同步控制部分不再像顺序环境中的 case 语句，而必需定义方法的调用条件，即"有条件的 case 语句"。在这种情况下，当子类中有新方法引入时，可能会改变从父类中继承过来的方法的调用条件，因而必须对同步控制部分重新定义。对于不同的语言，由于采用的同步机制不同，就会不同程度地对父类中原本可以继承过来的方法进行重定义，这就会带来一系列的问题，即出现所谓的继承异常。

3. 继承异常

所谓继承异常，是指为了保证并发对象的完整性，在某些情况下必须在子类中重新定义父类中原来可以继承的方法。继承异常带来的后果就是破坏了面向对象方法的最大优点：信息隐藏（封装性）和代码重用。

下面举一个简单的例子来说明这个问题。在基于体（Bodies）的并发面向对象语言中，每个对象都有一个称为 body 的内部控制线程，该 body 就是前面所述消息接收控制体，其中的控制语句类似于 Ada 语言中的 select 语句。body 收到消息后激活相应的方法。

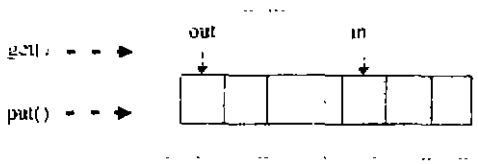

图2　有界缓冲区

如图2所示的先进先出（FIFO）的有界缓冲区类，它有两个公共方法 put()和 get()；put()插入一个元素到缓冲区中，get()从缓冲区中删除一个最先进来的元素。变量 in 和 out 指出当前要插入和删除元素的位置（in mod SIZE 和 out mod SIZE）。显然，当缓冲区为空的状态时，只能接收消息 put()，到达的 get()消息则不能接收处理，而只能保存在未处理消息队列中；当缓冲区为满的状态时，只能接收 get()消息；当缓冲区为非空非满状态时，既可接收 get()消息，也可接收 put()消息。其定义如下：

```
Class b-buf:ACTOR {//b-buf is an Actor
    int in, out ,buf[SIZE];
public:
    void b-buf(){in=out=0;}
    void process_put(){//store an item
        in++;
    }
    int process_get{//remove an item
        out++;
    }
    void body(){
        loop {
            select{
                accept get() when(!(in==out))start
                        process_get();
            or
                accept put() when(!(out=in+SIZE))
                        start process_put();
}}}}
```

图3　基于 body 的有界缓冲区对象的定义

当在其子类中引入新的方法如 empty()（判断缓冲区空否）时，其 body 部分必须要加入对该新方法 empty()的同步约束，否则就无法控制该新加消息的接收，因此对父类中 body 需要进行重定义，这就是前面所说的继承异常。实际上对于采用显式的消息接收机制（如 ABCL/1、CSSA）、路径表达式（如 Procol）、Direct Key Specifications（如 SINA、OTM）等机制的语言也会存在上述类似问题。

4. 继承异常现象以及解决方法

4.1 继承异常的表现

以上面的缓冲区为例子，为了解决上例中基于body机制的语言中所产生的继承异常现象，一种最直接的方法是引入对象状态的概念。根据对象可接收消息的不同(实际上就是同步约束条件的不同)将b-buf的对象分为三种状态：empty(空缓冲区状态，只能接收put消息)、partial(非空非满状态，put和get都可接收)、full(满缓冲区状态，只能接收get消息)。每个状态都关联着一个可接收消息集合。这样，当在子类中引入新方法(如empty)时，我们只需将新方法加入到相应对象状态的可接收消息集合里，而无需重新定义同步控制方法体。然而，并非对加入的任何方法都可以这样简单地解决继承异常问题，当某些特殊的方法加入时，由于其他方面的原因仍有可能需重定义父类中本来可继承的方法，产生继承异常。较典型的情况有以下三类：

4.1.1 对象状态的分解 当子类中引入新的方法使得对象可接收消息的状态进一步细分，从而引起子类中的同步控制与父类不同，就可能引起继承异常。比如，当在b-buf的子类x-buf2中加入方法get2()(同时取出缓冲区中的两个元素)时，由于调用get2()的约束条件为缓冲区中至少要有两个元素，如采用行为抽象(Behavior Abstraction)机制的语言，要将partial状态分为两种情况：x_one(缓冲区中只有一个元素的状态)，x_partial(缓冲区中有两个以上元素的状态)，此时子类x-buf2中的可接收消息集合有四部分：x_empty，x_one，x_partial，x_full。所以父类中的get()，put()方法体中的对象状态变化(become语句)应改变，使得父类中的方法get()，put()方法体在子类中需要重定义，从而产生继承异常。

4.1.2 与历史有关的可接收状态 新加入的方法的调用条件是与对象的历史消息有关，这种情况下也会引起继承异常。比如，在b-buf的子类gb-buf中增加新方法gget()，gget()除了不能在put()方法调用后立即执行外，与get()完全相同，这种约束条件需要另外增加变量来表示，并且在get()和put()方法调用后需要赋予它适当的值。因此这就会造成对父类中的get()和put()方法重新定义，引起继承异常。这里的gget()的调用条件就是与历史有关的。

4.1.3 对象状态的修改 新加入的方法使得其他方法的可接收状态只是原来的可接收状态的一部分。比如，混合类的例子，类lb-buf为Lock和b-buf的子类，类Lock起给对象加锁的作用，这时在lb-buf中的方法get()和put()的可接收状态比父类中的要少一半，也就是说，他们的调用条件与父类中的不一样。对于方法卫式(Method Guards)[6]机制，方法的调用条件是附着在方法体上的，因此一旦方法的调用条件改变(子类中的get和put调用条件比父类增加lock约束)，必然会产生方法的重定义。

4.2 原因分析

在并发面向对象的系统中，每个对象在接收消息后的处理可以分为以下三个过程：

(1)前置约束：根据同步约束条件判断该消息是否可接收，若可接收，则激活相应的方法；

(2)方法体的执行：执行整个方法体中的处理过程；

(3)后置转换：方法体执行结束后根据需要改变对象的状态。

显然，在这三个步骤中，(1)和(3)是与同步约束紧密相关的两个部分。继承异常产生的原因在于：这些语言中未能将这些有关同步约束的部分从类的方法体中彻底分离出来，子类中新方法的加入或对父类方法的修改有可能破坏父类中方法的同步控制条件，导致子类中重新定义父类中的方法，产生继承异常。因此，一个很自然的想法就是将并发对象中方法的同步控制代码与方法体的代码分开，对它们分别进行继承，应该说是一个解决继承异常问题的有效方法。

4.3 解决方法

继承异常作为并发面向对象语言中的一个很重要的问题，目前已有很多试图解决它的语言同步机制。根据其出发点不同，主要有以下两大类：

4.3.1 从同步代码与方法体分开的角度出发考虑 目前的绝大多数并发面向对象语言的同步机制都是出于这种思想，它们解决继承异常的程度取决于各自将同步代码与方法体分离的程度以及分离合理的程度。

基于行为抽象[5]的机制，它根据对象能接收的消息分为不同的抽象状态，当对象本身的状态发生变化时(实际上就是接收了消息)，相应的抽象状态也随着变化。不过，在这种机制下，抽象状态的变化是包含在方法体内的。也就是说，同步部分实际上并没有完全与方法体分开，因而不可避免地仍然会有继承异常产生。另一方面，抽象状态与具体状态之间

的变化的一致性问题也比较难以保证。使能集(Enable Set)机制是行为抽象机制的扩充，它对于解决大粒度的并发问题有一定的效果，但在小粒度并发系统中的开销却是相当大的(详细讨论见[8])。

基于方法卫式的机制，就是给每一个方法附一个卫式，当条件满足时可以接收该消息。这种方法由于将方法的同步约束与布尔表达式相关联，它对某些在行为抽象下引起的异常问题(如前面的 get2()异常)确实能提供一个很完美的解决方法；但由于该机制将条件与方法体没有能完全分开，再加上卫式表达式的局限性(无法直接表达那些与历史有关的信息)，因此仍不可避免地会产生异常问题。

日本学者 StaoshiMatsuoka 和 Akinori Yonezawa[6]提出的解决继承异常的方法就是以从方法体中分离同步代码为出发点而设计的。它以方法集合和卫式为基础，结合二者的优点，利用同步子和转换说明较好地解决了前面所述的几种继承异常的典型问题。然而，此方法的主要不足在于在类的定义中引入了较多的机制，使得类的定义比较烦琐，缺乏层次性。

上述各种方法也可以看作是从方法可以接收的正条件角度来考虑的。但由于父类无法预料子类中新方法的加入所可能产生的变化，往往会产生继承异常，这正是这种方法的局限性所在。因此，便有了下面的 Frølund 的思想。

4.3.2 从方法不能接收的条件来考虑 Illinois 大学的 Frølund 提出的一种简单的框架，主要着力于导出方法的同步代码的重用。这种同步机制是基于方法卫式的。它指出当某条件成立时相应的方法不能接收，即否定卫式。假设给定一方法 m，那么只有当该包含 m 的所有父类中有关该方法的卫式皆为 false 时才可以接收消息 m。因为如果一个方法在父类的某个状态下不可接收，显然该方法的导出方法在子类的该状态下也不可接收。这个否定卫式在类的继承链上逐步加强。

尽管该方法从反面来考虑，对于解决子类中的不可预料性有一定的效果，并且对于解决状态分解异常(采用卫式)和卫式的重用都有一定的成效，但它仍存在以下的不足：

①仅着眼于卫式的重用，对其他种类的继承异常没有给予足够的考虑(比如与历史有关的异常)。②这种机制还没有一个实现系统；因为如何有效地实现该思想存在一定的难度(卫式与方法不再有一一对应的简单关系)。

5. 一种处理继承异常的新方法

前面对并发对象环境中的继承问题进行了较详细的探讨，并总结了当前对该问题的一系列解决方法及其优缺点。实际上，上述方法在解决继承异常方面的基本思路是根据对继承异常现象的直观理解，将同步机制与方法体代码逐步分开，分别加以继承。我们认为，要从根本上解决继承异常，使得同步代码在更大范围内得以复用，首先应对继承异常现象进行深入的研究，找出其根本问题，在此基础上将同步代码与方法体完全分开，并且建立相互间的各种对应关系，对同步代码的复用提供全方位的支持。

所有解决继承异常的同步机制的提出都是从如何继承同步代码这个角度来考虑的，或者说，如何避免可继承方法的重定义以及如何解决同步代码的重用问题实际上是继承异常的本质问题。实际上，同步代码的重用可以有两种方式：一种是通过继承的方法而重用，它允许对同步代码作适当的修改；另一种则是直接复用父类中的同步代码。

我们认为，要对同步代码的复用提供全方位的支持，从根本上解决继承异常问题，必须解决好以下三方面的问题：

①同步代码的分离问题，将同步代码从方法体中分离出来是解决继承异常的最基本的要求，这可由前面的例子以及分析可以知道。

②同步代码的分解问题，由于子类中的方法与父类中的方法的同步控制约束有很大的相似性，但同时也会存在部分的不同，因此如果能将其中不变的那部分同步代码继承过来，将对同步代码的重用提供一个很好的手段，这里的分解问题实际上就是考虑如何合适地对同步代码进行模块化，使得子类能重用父类中没有变化的同步代码。

③同步代码的独立问题，实际问题中有很多情况的同步控制模式是完全一样的，如读-写模式等等，如果能将这种模式写成一个公共的模式，再把该同步模式关联(connecting)到具体的问题上，这样就可以使得同步控制代码更加灵活，可以在更大程度上达到重用同步代码的效果。

从代码重用的角度来看，上述三个问题分别是方法体的重用、部分同步代码的重用、整个同步代码的重用三个不同程度重用的问题。迄今，尚未见一种方法能同时满足这三方面的要求。基于上述考虑，我们提出了一种类的双层描述思想来解决同步代码的重用问题。它首先将类分成两层结构，将同步控制与

具体方法分开;以类的形式封装,便于同步代码的模块化和独立性;提供的绑定机制使得同步模式的重用具有更多的灵活性.因此,该方法满足了前面所说的分离问题、分解问题和独立问题三方面的要求,使得同步代码在更大的范围内达到重用的效果,比以前的方法具有更大的灵活性.关于该方法的具体讨论详见文[9]。

结语 解决并发面向对象语言中的继承异常是一个困难的问题.本文主要对现有的研究工作加以分析,并提出了一种解决此问题的新的思想和方法.进一步的工作在于将上述方法进一步的细化,并融入并发面向对象规约语言和并发面向对象程序设计语言之中.这部分内容将另文讨论.另一方面,目前关于并发面向对象中继承问题的讨论主要集中于被动对象类中同步代码的继承问题,而对于主动对象及其协同代码的继承问题的讨论未见涉及.值得开展进一步的讨论.

参 考 文 献

[1] 徐家福等.对象式程序设计语言.南京大学出版社,1992年

[2] Barbarab. Wyatt et al. Parallelism In Object-Oriented Languages: A Survey. IEEE Software, Nov. 1992

[3] Yasuhiko Yokote, et al, Concurrent Programming in Concurrent Smalltalk, In Object-Oriented Concurrent Programming, (ed) Akinori Yonezawa et al, MIT Press, 1987

[4] Yasuhiko Yokote, et al, POOL-T: A Parallel Object-Oriented Language, Same to [3]

[5] Dennis G. Kafura, et al, Inheritance in Actor based concurrent object-oriented languages, In Proc. of E-COOP'89, Cambridge University Press, 1989

[6] Staoshi Marsuoka, et al, Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, In Research Directions in Concurrent Object-Oriented Programming, (ed) Gul Agha, et al, MIT Press, 1993

[7] Gregory R. Andrews, et al, Concepts and Notations for Concurrent Programming, Computing Surveys, 15(1)1983

[8] Satoshi Marsuoka et. al, Highly Efficient and Encapsulated Re-sue of Sychronization Code in Concurrent Object-Oriented Languages, ACM SIGPLAN NOTICES, 28(10)1993

[9] 张鸣等.基于类层次结构的继承异常处理方法,第七届全国计算机青年工作者会议(上海 NCYCS'98)录用

(上接第39页)

同步/异步的互操作和阻塞/非阻塞的消息传递机制,只要在 KQML 消息中置上合适的参数即可。

KQML 可使用任何通讯协议作为自己的通讯机制(http,smtp,TCP/IP 等).由于 KQML 消息的内容是透明的,故其内容可用任一语言来书写.Facilitator 提供了在大型网络上发现知识的能力,能与其它 WWW 上发现知识的应用程序进行互操作。

关于 KQML 的安全性,应当做在传输协议层还是语言层仍没有定论.如果做在语言层,则应当引入新的行为原语和消息参数来对消息的内容或对整个消息加密[3]。

综上所述,KQML 是一种新型的 agent 通信语言,它能满足 agent 间通信的绝大部分需要.将来的发展是使其在工业界标准化并加上安全性能。

参 考 文 献

[1] Y. Labrou et al, A Proposal for a New KQML Specification. ARPA Knowledge Sharing Initiative, External Interfaces Working Group Working Paper, February 1997

[2] Y. Labrou et al, A Semantics approach for KQML-a general purpose communication language for software agents. Third Intl. Conf. on Information and Knowledge Management (CIKM'94), November 1994

[3] C. Thirunavukkarasu, et al, Secret Agents-A Security Architecture for the KQML Agent Communication Language. CIKM'95 Intelligent Information Agents Workshop, Baltimore, December 1995

[4] T. Finin, et al, KQML as an agent communication language. The Proc. of the 3rd Intl. Conf. on Information and Knowledge Management (CIKM'94), ACM Press, November 1994

[5] J. Mayfield, et al, Desiderata for agent communication languages. Proc. of the AAAI Symposium on Information Gathering from Heterogeneous, Distributed Environments, AAAI-95 Spring Symposium, Stanford University, Stanford, CA. March 1995