计算机科学2000Vol. 27№ 3

并发面向对象模型中继承异常的一种解决方案

An Approach to Solve Inheritance Anomaly in Concurrent Object-Oriented Model

徐梦峤 陈家骏 郑国梁

(南京大学软件新技术国家重点实验室 计算机科学与技术系 南京 210093)

Abstract Inheritance anomaly is an important and difficult problem in the process of combining concurrency with object-oriented technology. There are many researches on it recently. Among them, a proposal by Satoshi Matsuoka and Akinori Yonezawa integrated the advantages of message sets and guarded method, which provides a good method for solving the inheritance anomaly, but there are some drawbacks when solving the history-sensitive anomaly. In this paper, based on the state machine, we developed the proposal and proposed a method for solving inheritance anomaly based on history-state to deal with the history-sensitive inheritance anomaly better.

Keywords Concurrent object-oriented, Inheritance anomaly, Concurrency constraint, History-state, State machine

1 引言

继承机制是面向对象软件开发中的重要设施,是实现软件复用和扩充的一种有效的语言机制,它是顺序面向对象语言的一个基本特点[7]。但是,在将面向对象技术与并发相结合的过程中,对并发控制机制处理不当,将会引起继承机制与并发控制间的冲突问题。这种冲突主要表现在:定义子类时需要修改(重定义)父类中的所有代码才能实现自身的并发控制,从而使得子类无法继承父类的代码,产生继承异常现象。继承异常的出现将进一步破坏面向对象技术的两大优点,即:继承和封装。所以,如何有效地避免继承异常或将继承异常的产生降低到最小的程度是并发面向对象模型所要研究的一个重要问题。

目前国际上对继承异常的研究很多,诸如 ECOOP、OOPSLA 等一些重大的面向对象技术国际性学术会议,近年来有大量的论文讨论继承异常问题。在文[1]中分析了产生继承异常的三种典型现象:可接收状态的分解产生的异常、与历史相关的可接收状态产生的异常和可接收状态的修改产生的异常。针对这三种现象,不少研究者从同步代码与方法体分离等不同角度提出了多种不同的避免继承异常或降低继承异常产生可能性的方案[2~6]。但大多方案只能部分解决问题,例如文[1]对于与历史状态相关的可接收状态的情况所涉及的与多步历史状态有关的继承异常现象,并未提出有效的解决方案。本文从这个问题出发,通过对继承异常现象的深入分析,借鉴了状态自动机中的思想,对上述方法进行改进,提出一种基于历史状态的并发对象描述来解决继承异常的方法,较好地解决了多步历史状态相关的继承异常现象。

2 继承异常现象及现有解决方案分析

2.1 继承异常

在面向对象中引入并发后,对象间的消息传递就不再具有顺序面向对象环境中的单一性和确定性。一个完整的并发对象必须包括两个部分:并发控制和消息处理。但是,如果引入的并发控制机制不恰当,将造成在定义子类时,需要修改父类的并发控制,这往往又会涉及到对父类的消息处理方法进行重定义,严重时,往往要对父类的每个方法都要进行,从而使得子类无法继承父类的代码,产生继承异常现象。

通过对继承异常现象的深入分析,发现其产生一般有三种比较典型的情况[8]:由接收状态分解产生的继承异常、由与历史相关状态产生的继承异常及由接收状态修改产生的继承异常。我们认为,继承异常产生的原因是,在并发面向对象系统中,对象的消息处理的两个主要部分(同步约束描述代码部分(即对象的并发控制)和消息处理的功能性代码部分(即对象的方法)),未能彻底分离,致使子类中新方法的加入或对父类方法的修改都有可能要求父类中的同步控制条件加以改变,若父类中的方法不经过重新定义就不能用于子类。

目前世界上对继承异常问题的研究很多，不同的学者从不同的出发点提出了各种各样的解决方法。其中有基于行为抽象的方案[2]，基于可操作集合的方案[3]，基于方法卫式的方案[4]，及基于重写逻辑的形式化方案[5]或将几种方法结合使用的混合方案[1]等。这些方案都能在一定程度上解决继承异常，但是都各有其缺陷。如基于方法卫式的解决方案虽然能避免状态分解异常并允许卫式在某种程度上的复用，但是它并没有考虑避免其他异常且无明显有效的实现方案；而基于重写逻辑的形式化方案虽然也可避免状态分解异常且重写规则灵活，但它也未对其他类型的异常进行研究，并且在细粒度的情况下实际应用效率不高。Satoshi Matsuoka 等结合基于方法卫式的方案和基于接收集合的方案在1993年提出了一种综合的方法来解决继承异常问题，达到了较好的效果。但该方法对于解决与历史相关的继承异常仍存在缺陷，它只能处理一次历史状态的情况。本文下面将着重就该方法作一介绍。

2.2 Satoshi Matsuoka 等的方案

该方法所提出的并发对象模型是 ABCM 的一个扩充。对象在活动状态时收到的消息放入消息队列中等候处理。消息处理的过程为：首先计算同步控制集合，决定当前状态下可接收的消息；搜索消息队列中可接收的消息，执行相应的方法；执行结束后，再执行有关该方法的转换规约，修改转换规约中指定的可接收集合和方法集合。

在解决继承异常时，文[1]将基于卫式和基于接收集合的同步控制两者相结合，重用同步代码的方式和顺序面向对象技术中引用父类方法类似。其中，方法集合(method sets，即方法名与相应方法体结合在一起的集合)是该方案中的一个基本概念。方法集合在子类中可以继承，也可以重新定义。

方案中使用同步器或转换规约(transition specifications)表达子类中的状态分解避免状态分解异常的产生，而对于状态修改，同步器亦可重定义或替换。转换规约用作同步器的可选机制。某一方法的转换在方法体执行完成后立即执行。它的作用在于说明一个对象接收集合的转换行为，反映一个对象内部状态所表示的同步约束。同步器和转换规约的定义形式可见文[1]。

该方案可以使用三种方式避免继承异常的产生：

1)使用同步器避免状态分解异常的产生。2)转换规约有两种方法表达子类中的状态分解：一是随分解需要增加转换集合，即仅新增方法的子类状态分片，其余自动复用；二是动用动态计算的卫式方法集合，即只需对与卫式相关的方法集合作重定义。3)对状态修改，同步器可重定义或替换、转换可根据状态有效地"转向"方法集合。

下面以一个常用的有界缓冲区的例子来说明如何解决继承异常的方案。

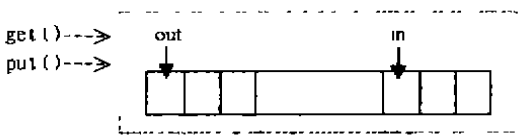

图1 先进先出有界缓冲区类

图1为一个先进先出(FIFO)的有界缓冲区类，put()和 get()是它的两个公共方法，其中：put()方法执行将一个元素放入缓冲区，get()方法执行从缓冲区中取出一个元素，变量 in 和 out 分别指出当前放入与取出元素的位置。

b-buf 的方法集合显然包括三种状态：当缓冲区为空状态时(EMPTY)，只能接收 put()消息；当缓冲区为满(FULL)时，只能接收 get()消息；当缓冲区为非空非满状态(PARTIAL)时，既可接收 put()消息，又可接收 get()消息。这样，该有界缓冲区类可定义成：

```
Class b-buf:{
    int size=in=out=0;int item[MAX_SIZE];
method_sets:
    mset EMPTY #{put}
        //对空缓冲区只能进行 put 操作
    mset FULL #{get}
        //对满缓冲区只能进行 get 操作
    mset PARTIAL EMPTY|FULL
        //非空非满缓冲区 put 和 get 操作均可执行
methods:
    void put(int item)
        {size--;out=(out+1)%max_size;return item
            [out];}
    int get(){size++;in=(in+1)%max_size;item[in]
        =x;}
transitions:
    transition default {//缺省的转换规约
        become EMPTY when(size==0);
        become FULL when(size==BUFSIZE);
        become PARTIAL otherwise;
    }
}
```

我们以该方案对历史相关状态继承异常的解决为例来说明，其余两种情况继承异常的解决可见文[1]，本文不再赘述。

如果在上述有界缓冲区类 b-buf 中加入一新方法 gget()形成子类 gb-buf，gget()方法基本与 get()方法相同，只是 gget()方法执行的前一操作不能是 put()操作。这样，gget 消息的可接收条件不能简单地用对象的内部实例来表示，必须增加额外的变量(如布尔量 after-put)才能表示出这种约束条件，那么，在子类 gb-buf 中必须对原来父类中的 put()和 get()方法进行重定义，以增加对新增变量(after-put)的赋值等语句，从

而产生继承异常。在文[1]中，新增的变量 after-put 放在方法集合中进行定义，而执行完一次方法体后 after-put 的赋值放在转换规约中进行，这样，也做到了方法体的继承，从而避免了继承异常的产生。

该方案注意了实用性意义，为同步约束编程提供了可选择的最好结构，方便了对面向对象程序设计有经验的用户的使用，提供了较高的封装度和同步代码的复用程度。但是，在我们的研究中发现，该方案对由与历史相关的状态引起的继承异常的解决尚有欠缺之处。在引入变量 after-put 后，由于该方案仅提供了一种 disable_once 的机制，因此仅能判断 get 前一次是否是 put 操作(即单步历史状态)并对之加以解决，而在程序的执行过程中，不可避免地需要判断前 n(n≥1)步历史状态并在此基础上施行后继操作。如在子类中引入 ggget()方法，其调用条件为其前两次操作必须均为 get()，其余与 get()方法完全一样。但该方法对这种情况下所可能产生的继承异常没有提出进一步的解决方案。所以，为了更好地避免一般情况下的继承异常，并将其产生的可能性减到最小的程度，我们在上述方法的基础上结合状态自动机的思想提出了一种基于历史状态的继承异常解决方案。

3 基于历史状态的解决方法

3.1 基本思想

Satoshi Matsuoka 等提出的方法仅提供了一个记住一次历史状态的机制(disable_once)，而没有象"disable_twice"之类的机制，因此无法处理当需要判断两次或多次历史状态的情况。通过对这类问题的分析，我们认为这种现象产生的本质是需要记住多次历史状态。因此，我们自然联想到形式化系统中与历史状态密切相关的状态自动机概念。在如何将这个概念的基本思想结合到解决与历史状态相关的继承异常的问题的分析中，我们对文[1]方案中的方法集合进行进一步的改写和扩充，将状态分成两类：一类为静态状态，即记录当前与对象的静态信息相关的状态(与原方案中的状态相同)；另一类为历史状态，记录对象与历史信息相关的状态。每个历史状态与一个方法集合相对应。在此基础上，我们提出了"状态集合"(state sets)的概念，即历史状态名与其对应方法的集合，它由两部分组成，即静态状态部分(用 sstate 标记)和历史状态(用 hstate 标记)部分。其基本定义形式为：

```
state_sets:
    sstate name#{method name,…};
        //静态状态表示
    hstate name#{method name,…};
        //历史状态表示
```

该定义表示：当对象处于所定义状态名的状态时，可以接收方法集合里的消息，并按指定方法名所定义的方法体来处理消息。状态集合可以被子类继承，也可以重新定义。

在这种情况下，任何一个对象在任一时刻可能同时处于多个状态，包括静态对象状态以及新的历史状态。其中，历史状态是标记历史信息的状态，使用它可以方便地表达有关方法调用的历史约束条件。

所谓与历史状态相关的继承异常已在上文定义，这里给出一个一般化的定义，即：并发环境中，方法的同步约束条件是与前 n(n≥1)步的状态有关的，例如：gget 方法的调用条件为其前一操作不能为 put，ggget 方法的调用条件为其前两次操作均不能为 put。

状态自动机思想的引入，大大简化了对历史状态的记录与判断工作。它不象路径表达式方法那样，需要写出所有可能的路径表达式以避免可能产生的继承异常，而是在初始状态的基础上，按每次接收的消息判断该进入以下某一状态，并以每一个确定的状态来记录所有从初始阶段开始的历史消息。如我们以{get，get，get}为例，我们可以用状态自动机将之表示成：

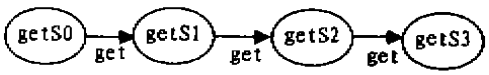

图2 状态转换图

其中，getS0为初始状态，getS1状态记录历史信息{get}，getS2状态记录历史信息{get，get}，getS3状态记录历史信息{get，get，get}。这样即可实现将所有消息接收的历史信息均记录于状态中。若当前消息为 gget，则通过对状态中所含历史信息判断，其前一步为 get 操作，所以 gget 消息可被接收；若当前消息为 ggget，同理通过判断，ggget 消息也可被接收。这样，无论对历史状态作怎样的限制，本方案都可以用上述方法来避免继承异常的产生。

本方案的消息处理机制为：如果对象同时处于静态状态 S1和历史状态 H1时，此时对象可接收的消息集合为状态 S1的可接收消息集合与状态 H1的可接收消息集合的交集。即：M(S1∧H1)=M(S1)∧M(H1)。其他有关消息的处理机制与原来的方法完全相同。如在先进先出有界缓冲区类中，基本静态状态有三个：EMPTY，FULL 和 PARTIAL。当 S1=EMPTY，H1=GETS0时，M(EMPTY)={put}，M(GETS0)={get，put}，所以有 M(EMPTY)∧M(GETS0)={put}，这一对应关系说明，在缓冲区处于空状态并且处于 GETS0状态的时候，只能 put 消息可以被接收。同理可得，在缓冲区处于非空非满状态并且处于 GETS0状态的时候，get 消息和 put 消息均可接收；在缓冲区处于

非空非满状态并且处于GETS1状态的时候，可以接收的消息集合为{get，put，gget}等等。

下面我们仍将通过具体的缓冲区的例子来说明我们的解决方案。这里先进先出有界缓冲区类 b-buf 的定义与第2.2节基本相同，只是 method-sets 被替换成 State-sets。在 b-buf 中引入 gget() 方法后，其状态转换图可表示为图3：

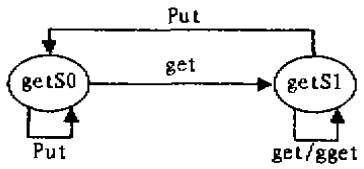

图3

子类 gb-buf 的定义为：

```
Class gb-buf:b-buf{
state_sets:
   //加入两个历史状态:GETS0,GETS1
   hstate GETS0        super PARTIAL;
   hstate GETS1        #{gget}|GETS0;
   sstate FULL         super FULL|GETS1;
methods:
    int gget(){return super get();}
transitions:
   //对 GETS0初始化
   transition put(){become GETS0;}
   transition get(){become GETS1;}
   transition gget(){become GETS1;}
}
```

在本例中，采用了 GETS0和 GETS1两个状态来记录历史信息，其中 GETS0表示前一次操作非 get，在此状态下可能接收的消息为 put，get；GETS1表示前一次操作为 get，在此状态下可接收 gget 操作或 GETS0状态下的操作，即其可接收的消息集合为{put，get，gget}，此时转换规约可定义成：接收到 put 消息后，历史状态转换为 GETS0状态；接收到 get 消息后，历史状态转换为 GETS1状态。这样，对于方法体本身，我们只需加入 gget() 方法的内容而不须进行重新定义父类中的 put() 和 get() 方法，从而避免了继承异常的产生。

结束语 将本文所提方案与基于行为抽象的方案[2]以及基于可操作集合的方案[3]进行比较，共同点在于三者均以不同的方式定义了一个对应于方法的集合。如基于行为抽象方案中的行为集合，基于可操作集合方案中的可操作集合和本方案中的状态集合，这些集合在一定意义上都表示了对象的状态，并指明对象在一定状态下可执行的操作。但是，本方案与后两者的最大区别在于两个方面：一是本文引入了状态自动机的思想，将对象的状态与历史信息紧密相连，从而可以使消息在处理过程中兼顾到历史状态，使子类中加入新的与历史状态有关的方法时所产生的继承异常得到很好的解决。二是本方案将状态的转换与方法体分离，更好地避免了其余两种方法中可能会产生的由状态分解所产生的继承异常。

澳大利亚学者 Lobel Crnogorac 等从行为类型的角度对继承异常问题进行了形式化的分析[5]，在并发环境中，由于需要引入并发控制，类型被定义为一系列方法序列的集合，但是每一条子类型的链却未必可以由一条增量继承层次的路径来实现，这就产生我们通常意义上的继承异常现象。从这个定义中我们可以看出，继承异常是与类型定义有关的概念[5]。结合对类型概念的非形式化与形式化两方面的理解，可以说，只要某种继承机制关于类型的定义来说是有异常的，那么不可能存在一种方法能完全消除继承异常现象。我们所作的工作只是设法提供必要的灵活性以减少代码重定义的频率即继承异常现象产生的频率及使其发生的程度达到最小。

参考文献

1 Matsuoka S, Yonezawa A. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In Research Directions in Concurrent Object-Oriented Programming. MIT Press, 1993. 107～150

2 Kafura D G, Lee K H. Inheritance in Actor Based Concurrent Object-Oriented Languages. In: Prec. of ECOOP'89. Cambridge University Press, 1989. 131～145

3 Tomlinson C, Singh V. Inheritance and Synchronization with Enabled-Sets. In: Proc. of OOPSLA'89, volume 24. SIGPLAN Notices, ACM Press, 1989. 103～112

4 Decouchant D, et al. A Synchronization Mechanism for Typed Objects in A Distributed System. In: Proc. of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrnet Programming, volume 24. SIGPLAN Notices, ACM Press, 1989. 105～107

5 Crnogorac L, et al. Classifying Inheritance Mechanisms in Concurrent Object-Oriented Programming. In: Proc. of ECOOP'98, Brussels, Belgium, 1998. 571～600

6 Meseguer J. Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. In: Proc. of ECOOP'93. 1993. 220～245

7 徐家福，等. 对象式程序设计语言. 南京大学出版社，1992

8 张鸣，等. 并发面向对象语言的继承异常问题研究. 计算机科学，1998，25(6)：14～18