基于双层类结构的继承异常处理方法*

张 鸣 吕 建 杨大军 陶先平

(南京大学计算机软件新技术国家重点实验室 南京 210093)
(南京大学计算机软件研究所 南京 210093)
E-mail: lj@netra.nju.edu.cn

摘要 继承异常是并发面向对象语言中的一个很重要而且也是处理起来很困难的问题,它会使同步代码难以重用.为解决此问题,文章提出了一种新的双层类结构模型的方法,来解决继承异常问题,它可对同步代码的复用提供全方位的支持.

关键词 继承异常,控制类,具体类,关联机制.
中图法分类号 TP311

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(inheritance)是顺序面向对象语言(sequential object-oriented languages)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(reuse)和可扩充的有效语言机制.

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(process)、管程(monitor)等,Smalltalk[2]中采用的就是这种方法.但由于对象与进程概念的不同,会在问题建模和进行对象保护时产生问题.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采用的就是这种方法.对象既是一个自包含模块,又是一个并发执行单位.无论采用何种途径,面向对象技术与并发机制结合都有一个重要问题——继承问题,即如何在并发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[3],COOL,Presto,Emerald等[4].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.另一途径就是如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,但许多工作都需由程序员来完成.

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语言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.本文首先分析了继承异常问题,然后提出一种处理继承异常的新方法,并通过例子加以说明.

1 继承异常及解决途径分析

所谓继承异常就是指,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

* 本文研究得到国家自然科学基金、国家杰出青年科学基金和国家攀登计划基金资助.作者张鸣,1973年生,博士生,主要研究领域为面向对象语言,形式化方法.吕建,1960年生,博士,教授,博士生导师,主要研究领域为软件自动化,并行程序形式化方法,并发面向对象语言和环境等.杨大军,1972年生,博士生,主要研究领域为面向对象语言,形式化方法.陶先平,1970年生,讲师,主要研究领域为软件体系结构,面向对象应用框架,软件自动化.

本文通讯联系人:张鸣,南京 210093,南京大学计算机软件研究所

本文1998-04-03收到原稿,1998-06-22收到修改稿

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.

根据不同的同步控制机制在解决继承异常时所面临的问题,文献[5]中以有界缓冲区为例说明了继承异常的3种典型情况:状态分解产生的异常、历史有关状态产生的异常、状态修改产生的异常.通过分析可以看出,继承异常产生的原因是没能将同步约束代码与方法体代码从根本上分开,因而导致在子类进行继承时无法继承本来可以继承的父类中方法的代码,产生继承异常.

继承异常作为并发面向对象语言中的一个很重要的问题,目前已有许多试图解决它的语言同步机制.从分离同步约束代码与方法体代码的角度(方法可以接收的正条件)出发,有Behavior Abstraction[6],Method Guards以及Synchronizers和Transition Specifications[5,7]等方法;从方法不能接收的条件角度考虑,比较典型的就是Illinois大学的Frfund提出的一种基于Method-Guards的框架[8].上述方法虽有各自的特点,但尚未有一种比较理想的方法来简洁地处理继承异常问题,提供对同步代码重用的全方位的支持.

归结起来,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3方面的问题:

• 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,它可避免同步代码与方法代码在继承中的相干性.

• 同步代码的分解问题 虽然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,但同时也存在部分不同.因此,如果能将其中不变的那部分同步代码继承过来,将会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.

• 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.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)到各个具体的问题上,就可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3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3个不同程度重用的问题.或者说,前两个问题是从继承的角度来考虑代码重用,第3个问题则是对同步代码加以直接复用.迄今为止,尚未见到哪一种方法能同时满足这3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.

2 基于类层次结构的方法

为了解决分离问题,我们首先将同步控制部分与类体部分分开,从而引入了控制类与具体类的概念.为了解决分解问题,更大程度地支持同步代码的重用,我们对并发环境中同步控制部分的执行特点以及对类本身产生的影响稍加分析就可以发现,同步控制实际上可以分成两个子部分:一部分是前置条件判断,即在接收到消息时的判断;另一部分是后置状态转换,即具体方法执行结束对抽象状态的影响.因此,一个完整的控制类的定义应该包括以上两部分的控制.基于上述考虑,我们引入了guard机制和消息集合相结合的方法,从而可以解决较小粒度的重用问题.为了解决独立问题,我们引入了关联(connecting)机制,它使得控制类可以灵活地与其他各类不同的具体类相连接.

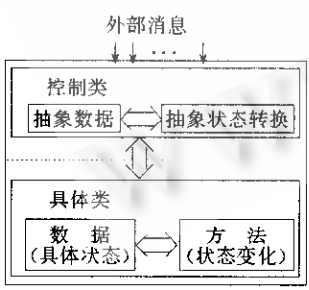

图1 类结构简图

在并发环境中,我们可以认为类由两个子类构成,分别是控制类和具体类.其结构如图1所示.

控制类描述类的抽象状态及其转换关系(或说是操作),其主要作用是完成类的消息接收的同步控制,主要有抽象数据(抽象状态和抽象变量)和抽象方法.具体类则是指描述具体状态及其转换关系的部分,或者说就是具体的实例数据及其上的操作.实际上,这一部分与顺序环境中的完全一样.

抽象状态是控制类中最核心的数据,是用来表示对象当前所处的不同消息控制状态,与消息的接收有关.也就是说,在不同的抽象状态下,类所能接收的消息不一样.抽象状态是由控制类中的抽象变量来定义的.引入抽

象变量的目的在于,隔离控制类与具体类之间的直接依赖关系.这一点在下面将会有较具体的说明.

控制类和具体类两者之间具有相对的独立性.他们之间的关系是通过特定的关联关系来进行连接.对象在接收到消息后,首先根据对象当前所处的抽象状态以及关联关系来确定该消息是否可接收.若可接收,则执行相应的具体类中的方法体,在结束后,如果控制类中还有相应的状态转换(behavior)部分,则执行之.

由于控制类也是一种类,因而它也有继承等有关类的特性.控制类与并发环境中对象的同步控制有着密切的关系,因此,本文将着重讨论控制类的设计.

3　语言表示机制

3.1　控制类的定义

抽象状态是控制类中最核心的数据.控制类主要就是围绕抽象状态来进行工作的.一般来说,控制类的定义需解决3个问题:(1) 抽象状态的定义;(2) 抽象状态与可接收消息集合的对应问题;(3) 抽象状态与具体状态变化的一致性.

(1) 抽象状态的定义方法

抽象状态采用表达式来定义:*state_name*:=*expression*,其中 *expression* 是用抽象变量来定义的.根据抽象变量与具体类数据能否建立关联,可以有以下两类情况:

(a) 可以与具体状态建立直接关联的 *expression*.在这种情况下,抽象变量能与具体变量建立直接对应关系.也就是说,抽象状态的变化依赖于具体状态的变化.实际上,这隐含了抽象状态与具体状态之间的映射关系,从而使抽象状态的变化可随着具体状态的变化而自动地进行.

(b) 不能与具体状态建立直接关联的 *expression*.抽象状态的定义与具体状态无关.需要定义控制类的变量.为了保证其变化与具体类之间的一致性,需要在具体类的方法执行完毕后由控制类修改相应的抽象变量,以保证抽象状态与具体状态的一致性.

(2) 抽象状态与可接收消息集合的对应问题

采用方法集合来定义状态的可接收消息:$M(state\_name)\equiv method\_set$;

通过建立抽象状态 *state_name* 与消息集合 *method_set* 的对应关系,来表示当对象处于抽象状态 *state_name* 时,可以接收集合 *method_set* 内包含的消息.由于 *method_set* 是以集合的形式来定义,因此可以有相应的交、并等基本集合运算,便于重用.这里的消息(方法)集合是指抽象方法集合,它们与具体类方法之间的关联关系则由关联机制来实现.

(3) 抽象状态的转换关系

主要解决与具体类状态转换之间的一致性问题.

通过定义一个抽象状态转换部分 behavior 来完成上述功能.

method_set⇒*expressions*,表示在 *method_set* 中的方法执行结束后还需完成的动作.该部分不是必需的.如果在抽象状态的定义中包含了无法与具体状态建立直接关联的部分,则为了保证二者的变化一致性,需要定义这一部分.

3.2　控制类与具体类之间的关系

系统中任一对象 O 接收到某消息 m 后,将消息 m 存入该对象的消息等待对列,然后按下述机制进行消息处理过程:

(1) 首先扫描本对象的消息等待队列,若队列为空,则没有待处理的消息,处于空闲状态;否则,选取队列的第1个消息 m,继续步骤(2).

(2) 根据 O 当前所处的抽象状态,决定 m 是否为可接收消息.若不可接收,则消息 m 继续在等待队列待处理;若可接收,则将 m 送出等待队列,继续步骤(3).

(3) 执行相应的具体类的方法 m 的方法体部分.

(4) 控制类若有与 m 相应的 *behavior* 部分的转换,则在执行完步骤(3)之后继续执行该相应的部分.

实际上,控制类定义的抽象状态组成一个或多个状态变化循环序列.同一状态循环序列内的各个状态之间是互斥的.也就是说,任一对象在任一时刻不可能处于一个循环中的不同状态(这就保证了消息同步控制的无歧义性),但可以处于不同循环中的不同状态,只要消息属于其中一个状态的可接收消息集即为可接收消息.实际上,不同状态循环只是从不同角度来描述对象的状态变化过程及其特点.

3.3 关联机制

为了能使同步控制满足前面所说的分离性和独立性两方面的要求,根据上面对控制类所需解决问题的分析,采用所谓的关联机制来处理这个问题.

由于控制类采用上述方法来定义,因此,为了能将控制类的同步控制施加到具体的类上,必须在控制类与具体类之间建立一种特殊的关联关系,以保证两者之间的一致性.这种关系应包含两方面的内容:(1) 抽象变量与具体变量之间的关系,(2) 抽象方法(集)与具体方法之间的关系.可以采用下面的形式建立关联:

connecting 〈control_class_name〉;

[by 〈关联关系定义〉 end],

其中〈关联关系定义〉详细刻画了抽象变量与具体变量以及抽象方法与具体方法之间的关联关系.

不论控制类如何定义,控制类与具体类之间的关联关系都是通过显式和隐式两种方法给出的.隐式的方法主要就是利用同名相关联的原理进行.显式定义就是在 connecting 语句之后明确给出抽象量与具体量、抽象方法与具体方法之间的关联关系.

· 显式定义:明确给出抽象变量与具体变量之间、抽象方法(集)与具体方法之间的关联关系.

· 隐式定义:针对一些特殊的情况,如果抽象变量与具体变量、抽象方法(集)与具体方法同名,则隐含地认为二者之间有直接的关联关系.

该机制使得抽象的同步控制部分可以十分灵活地用于不同的类上,尤其适合于那些通用的同步控制模式,提高了同步代码的重用程度.

3.4 控制类的继承

由于控制类是一种类,因而具有类的很多特性.其中,继承性是很重要,也是对重用有很大作用的特性之一.

子类可以通过继承机制继承父控制类中的数据.控制类继承的时候一般来说有两种情况:一种是没有发生抽象状态变化的继承,另一种就是抽象状态发生变化的继承.第 1 种情况很简单,最多就是改变一下抽象状态与抽象方法的对应关系.这里不作详细的讨论.下面将着重说明抽象状态发生变化时的处理问题.

一般来说,子类继承父类后对父类中状态的变化有两种情况:① 子类对父类状态的分解,② 子类中增加一些新的状态.因此,相应地引入 decompose 和 add 两种方法来处理.

(1) 基于状态继承的 decompose 方法

对于状态分解异常,可以采用 decompose 机制来处理.

decompose *state* into 〈*state_list*〉;

该机制主要针对状态分解产生的继承异常问题.将需要分解的父类中的抽象状态 *state* 通过 decompose 在子类中分解为多个状态〈*state_list*〉,其他不需改变的状态则直接继承过来(子类中也就不存在抽象状态 *state*).对于相应状态的可接收消息集合也作相应的重定义(能继承的则继承过来).对父类中消息集合的引用采用 *super*.* 进行.

(2) 基于状态继承的 add 方法

该机制就是在子类中增加新的抽象状态,原来父类中的状态则以继承的方式继承过来,仍然有效.

add 〈*state_list*〉;

表示子类中将增加〈*state_list*〉中的新的状态.父类中的状态被子类继承过来仍然有效.此时,原来父类中关于抽象状态的可接收消息集的定义可以继承过来,但在子类中需增加对新增抽象状态的可接收消息集的定义.

对于不符合上述两种方法的情况,则采用一般的定义方法.

4 例　子

下面,我们就用前面所介绍的方法来处理先入先出(FIFO)的有界缓冲区例子所产生的几种典型的问题.有界缓冲区对象有两个公共方法 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.*size* 表示当前缓冲区中元素的数目,MAX 表示缓冲区最大容量.显然,在并发环境中,对 get 和 put 方法的调用需要相应的同步约束条件.利用前面所介绍的方法可定义缓冲区对象如下:

```
control class   buffer_sync {                    //控制类定义
        state   empty, partial, full;             //抽象状态
        int     size,MAX;                         //抽象变量
        method  get,put;                          //抽象方法(集)
        relation
                empty:=(size==0);
                partial:=(0<size<MAX);
                full:=(size==MAX);    //可接收消息集的定义
                M(empty)≡{put};
                M(partial)≡{get,put};
                M(full)≡{get}
}
class   buffer  {                                 //具体类定义
    connecting  buffer_sync                       //隐式关联控制类 buffer_sync
        int     size,buf[MAX],...;
    public
        int     get() {...                        //取走一元素
                    size--;  }
        void    put() {...                        //插入一元素
                    size++;  }
}
```

4.1 get2引起的继承异常

当在类 *buffer* 的子类 *x-buffer* 中加入新方法 get2()(同时取走缓冲区中的两个元素)时,会引起状态分解异常.在这里,采用 decompose 机制解决如下:

```
control clas        x-buffer_sync:buffer_sync  {  //继承 buffer_sync 的同步代码
        state                                     //empty,full 继承父类;
          decompose partial  into x-one,x-partial;//分解状态 partial
        method      get2;                         //抽象方法,get,put 继承过来
        relation
            x-one:=(size==1);
            x-partial:=(1<size<MAX);
            '//'可接收消息集的定义,'|'为集合的并操作
            M(x-one)≡super.M(partial);
            M(x-partial)≡{get2}||super.M(partial);
            M(full)≡{get2}||super.M(full)
}
class               x-buffer:buffer  {            //具体类定义
    connecting      x-buffer_sync                 //隐式关联控制类 x-buffer_sync
        public                                    //实例变量、get(),put()继承父类;
        get2()   {...                             //取走两个元素
                 size:=size-2;    }
}
```

4.2 gget 引起的继承异常

当在类 *buffer* 的子类 *x-buffer2*中引入新方法 gget()(除了不能在 put()方法调用后立即执行外,与 get()完全相同)后,会产生与历史有关的继承异常.这里应用 Add 机制可按如下定义解决此问题:

```
control class       x-buffer2_sync:buffer_sync {  //控制类定义
        state                                     //empty,partial,full 继承父类;
```

```
        add     gget-stat;                    //增加新状态 gget-stat
      bool      after-put;                    //抽象变量,其他继承过来
      method    gget;                         //抽象方法;其他继承过来
      relation
          gget-stat:=(after-put==false) && super.partial;
          M(gget-stat)≡{gget};
      behavior                                //抽象状态转换部分
          {put}⇒{after-put==true;}            //put 继承父类中
          default⇒{after-put==false;}         //缺省所作的状态变化
}
class             x-buffer2:buffer  {       //具体类定义
      connecting  x-buffer2-sync              //隐式关联控制类 x-buffer2-sync
      public                                  //实例变量、get(),put()继承父类;
             gget(){super.get()}
}
```

4.3 关联机制的例子

下面以读者和写者模式为例来说明关联机制. 写操作之间、写操作与读操作之间必须互斥;读操作之间可以同时进行(当然,该功能也可以通过其他方法来实现,这里仅说明关联机制的特点).

R_op 和 *W_op* 分别代表读者类和写者类的方法. 可以定义这样3种抽象状态:*idle_state*,*read_state*,*write_state*. 状态 *idle_state* 表示当前没有任何方法在执行,可以接收读和写操作;状态 *read_state* 表示当前已有读操作在执行,因此为保证互斥性,只能接收读操作,而不能接收写操作;状态 *write_state* 表示当前正在进行写操作,拒绝任何其他操作. 采用计数器(抽象变量)来表示当前读、写操作的情况. 具体定义如下:

```
control class    reader-writer  {          //控制类的定义
      state      idle-state,read-state,write-state;
      int        r-count,w-count;          //抽象变量
      method
          R-op   {r-count++;}              //抽象方法操作定义
          W-op   {w-count++;}
      relation
          idle-state:=((r-count==0) && (w-count==0));
          read-state:=((w-count==0) && (r-count!=0));
          write-state:=(w-count==1);
          M(idle-state)≡R-op‖W-op;
          M(read-state)≡R-op;
          M(write-state)≡{ };
      behavior                             //状态转换
          R-op⇒{r-count--;}
          W-op⇒{w-count--;}
}
```

下面,我们希望定义这样一个缓冲区的类,其中有两类方法:一类是关于测试缓冲区状态的方法,如 empty(缓冲区空否)等;另一类是改变缓冲区内部元素的方法,如 get,put 等. 现在要求这两类方法按读-写模式进行同步控制,即第1类方法相当于读-写模式中的"读"操作,第2类操作相当于读-写模式中的"写"操作. 应用关联机制定义如下:

```
class rw-buffer:buffer  {              //是类 buffer 的子类
      connecting reader writer         //显式关联控制类 reader-writer
        by  R-op⇔{empty};
            W-op⇔{get,put};
        end;
        public                         //get(),put()继承父类;
          empty(){                     //测缓冲区空否
                if (size==0) return true;
                else         return false;}
```

5 结 语

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,必须解决好同步代码的分离、分解和独立这3个方面的问题.迄今为止,尚未见到有哪一种方法能同时满足这3方面的要求.因此,我们提出了上述类的双层描述思想来解决同步代码的重用问题,对解决继承异常问题进行了新的尝试.

由上面的例子我们可以看到,控制类的定义与具体类之间完全隔离于,一方面可以使具体类部分在继承的时候不受控制类的干扰,另一方面也可以使抽象的同步控制代码灵活地与其他具体的类相结合,不受具体类代码的限制.从而可以提供方法体、部分同步代码以及整个同步代码的3个不同程度的重用支持.

以上着重考虑的是不同对象间的并发以及对象内不同操作间的并发问题(不同对象或同一对象同时调用相同的方法),关于同一对象里操作内部的并发问题不是本文讨论的范围.对于同步点在操作中间的并发问题,则可以归结为对象内操作间的并发问题来考虑.

进一步的工作就是将上述思想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,并探讨其实现技术.另一方面,目前关于并发面向对象中的继承问题的讨论主要集中在被动对象类中同步代码的继承问题上,而对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:南京大学出版社,1992
(Xu Jia-fu *et al*. Object-oriented Programming Languages. Nanjing: Nanjing University Press, 1992)

2 Yokote Yasuhiko, Tokoro Mario. Concurrent programming in concurrent Smalltalk. Object-oriented Concurrent Programming. Cambridge: MIT Press, 1987

3 America Pierre. POOL-T: a parallel object-oriented language. Object-oriented Concurrent Programming. Cambridge, MA: MIT Press, 1987

4 Wyatt B *et al*. Parallelism in object-oriented languages: a survey. IEEE Software, 1992,9(11):56~67

5 Matsuoka Staoshi, Yonezawa Akinori. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In: Agha G, Wegner G, Yonezawa Akinori eds. Research Directions in Concurrent Object-oriented Programming. Cambridge, MA: MIT Press, 1993

6 Kafura D G, Lee K H. Inheritance in actor based concurrent object-oriented languages. In: Cook S ed. Proceedings of the European Conference'89 on Object-oriented Programming. Cambridge: Cambridge University Press, 1989

7 Matsuoka Satoshi *et al*. Highly efficient and encapsulated reuse of synchronization code in concurrent object-oriented languages. ACM Sigplan Notices, 1993,28(10):109~126

8 Frflund S. Inheritance of synchronization constraints in concurrent object-oriented concurrent languages. In: Madsen O L ed. Proceedings of the European Conference'92 on Object-oriented Programming. Berlin, Heidelberg: Springer-Verlag, 1992. 185~196

A Two-layered-class Method for Solving the Inheritance Anomaly

ZHANG Ming LÜ Jian YANG Da-jun TAO Xian-ping

(*State Key Laboratory for Novel Software Technology Nanjing University Nanjing* 210093)
(*Institute of Computer Software Nanjing University Nanjing* 210093)

Abstract In concurrent object-oriented languages, the inheritance anomaly is an important and difficult problem, which makes synchronization codes difficult to reuse. Based on the two-layered-class model, a new method for solving the inheritance anomaly is proposed. It can provide the flexible and sufficient support to the reuse of synchronization codes.

Key words Inheritance anomaly, control class, concrete class, connecting mechanism.