C++的一种并发扩充方案*

陈家骏 赵建华 郑国梁

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

摘要 该文给出了一种对C++进行并发扩充的方案.它基于这样的并发面向对象模型:系统由一组自治的并发对象构成,对象可以有一个体,一旦对象被创建,对象体就开始执行;对象间采用同步消息传递,允许对象内部的并发;对象的并发控制分散在各方法的激励条件中.文章还给出了一种转换策略,把扩充的C++描述转换成C++描述,使之能被现有的C++编译器识别.转换中利用了某些多任务操作系统(如Windows 95)所提供的多线程和同步设施.

关键词 面向对象,并发,C++,并发扩充,转换.

中图法分类号 TP312

面向对象程序设计OOP(object-oriented programming)技术中的对象、对象的自治性以及对象间通过消息传递进行协作等特性是对客观活动的一种直接模拟,采用面向对象模型开发软件显得更自然,其良好的封装、数据抽象及继承等特性有利于软件的并发与维护.

并发是面向对象模型的一种潜在特性,它使得面向对象模型可以用于描述并发计算.然而,目前一些流行的面向对象程序设计语言(如C++[1])只提供了描述程序顺序执行的能力.即使有些语言(如Smalltalk[2])提供了描述程序并发执行的机制,但它们往往采用了一些传统的并发控制模式和设施,没有从对象模型本身的特点来考虑并发控制,从而造成与对象模型中一些概念(如封装、继承及自治等)相冲突.[3]如何在面向对象模型中引进并发机制,使其能与对象模型中一些现有的特性有机结合?如何在面向对象程序设计语言中提供描述系统并发行为的设施,使之能方便地描述并发计算?这些是很多人都在研究的问题.[3~8]

本文首先分析一些可能的并发面向对象模型,提出了基于并发对象概念的并发面向对象模型.然后给出一种对C++进行并发扩充的方案,使之能够描述我们所提出的并发对象模型.最后,介绍一种转换策略,把扩充的C++描述转换成C++表示,使之能够被现有的C++编译器识别.

1 并发面向对象模型

1.1 如何在对象模型中引进并发

顺序对象模型由一组被动的对象构成.系统的执行行为是,外界激活其某个对象的某个方法,该方法的执行中向系统中其他对象发送消息,从而激活这些对象的方法的执行.这里的消息发送就是过程调用,即消息发送者必须等到消息接受者的相应方法执行完毕,才能继续执行下去,整个系统只有一个执行线程(Thread).

如何让上述的顺序对象模型并发起来?目前存在两种典型的方案.

(1) 采用异步消息发送.即一个对象向另一个对象发送消息后,前者不必等待后者处理消息即可继续执行下去,后者对消息进行缓存.[8]

(2) 对象可以有一个体(Body),一旦对象被创建,这个体就开始执行.[5]

在方案(1)中,由异步消息发送产生新的执行线程,从而引起系统的并发.由于采用异步消息发送,必须提供今后如何取得返回值的设施,如future variables.[8]异步消息发送的不足之处在于:它使得程序的行为难以把握,不利于对程序的推理(Reasoning).方案(2)通过创建新对象产生新的执行线程,在对象体执行的同时,对象还可以接收和发送

---

* 作者陈家骏,1963年生,副教授,主要研究领域为软件开发环境,自然语言处理.赵建华,1971年生,博士生,主要研究领域为软件工程,实时系统.郑国梁,1937年生,教授,博士导师,主要研究领域为软件工程,软件开发环境.

本文通讯联系人:陈家骏,南京 210093,南京大学计算机软件新技术国家重点实验室

本文1997-04-07收到原稿,1997-07-25收到修改稿

消息,消息发送可采用同步方式,但应允许对象内部的并发,否则,极易产生死锁.

1.2 并发对象

在上述方案(1)的并发面向对象模型中,对象都处于被动的提供服务状态,而在客观世界中,对象不总是处于被动地位,它们中的一些除了能够通过接收消息提供服务外,还应该能做自己的事情.例如,对于经典的5个哲学家就餐问题,其中的餐桌对象管理着5把叉子,提供就餐服务,它处于被动状态;而哲学家对象则处于主动地位,一旦被创建,它就开始思考问题,然后向餐桌对象发送消息,请求就餐.当然,哲学家对象也可提供一些服务,如:让他人提交问题.因此,我们认为,方案(2)所给出的并发对象模型更适合于描述客观世界中的并发活动.

目前,基于方案(2)的并发面向对象模型中往往把对象分为两类:一类有对象体,称为主动对象;另一类没有对象体,称为被动对象.主动对象参与系统的并发操作,被动对象只作为其他对象的成员,局部于这些对象,不参与系统的并发操作.我们认为这样做易造成使用上的混乱,因为它要求使用者必须知道所使用的对象是主动的还是被动的,否则,若把某些被动对象当作主动对象来使用,将使这些对象面临环境对其方法的并发调用,而在设计这些对象时并未考虑并发,这样,就可能造成对象状态的混乱.

对象是一个高度自治的实体,它应具有自动的自我保护能力,在设计时,应考虑环境可能对其方法的并发调用,而不应由环境对其行为有过多的假设.因此,我们用并发对象的概念来描述并发环境中的对象,它们可以有体,也可以没有体,在设计它们时,都必须考虑环境可能对它们的并发操作,这样设计的对象才适合于各种环境.

1.3 对象的并发控制

处于并发环境中的对象,随时都会面临环境对其方法的并发调用,为了保证对象状态的一致性与完整性,必须要对对象方法的并发调用进行同步控制.同步控制可以由消息发送者提供,也可以由消息接受者提供.显然,若同步控制由消息发送者提供,当消息发送者由于种种原因没有进行同步控制时,将会产生对消息接受者方法的并发调用,造成消息接受者内部状态的不一致与不完整.因此,消息接受者本身应该提供同步控制.

通过分析,我们发现:一个对象的同步控制可以分成两类:条件同步和互斥.对象的条件同步是指对象在某种状态下只能接受某些消息.如:对于有界缓存(Bounded Buffer)对象,当缓存为空时,只能接受put消息而不能接受get消息;缓存满时,只能接受get消息而不能接受put消息.对象的互斥是指,当处于某种状态下的对象可以接受(处理)多个消息时,如:缓冲不空又不满,既能接受get消息又能接受put消息,由于这些消息的处理都有可能使用对象的成员变量,如果让它们同时执行,就存在共享变量(对象的成员变量)问题,为了保证对象状态的完整性和一致性,这些方法必须互斥地使用这些变量.

如何在并发对象类中描述对象的同步控制,一般存在两种方式:集中控制和分散控制.集中控制方式是把对象的同步控制放在对象(类)中某一个地方集中进行描述.例如:用对象体来控制对象何时接收消息就属于集中控制;另一种集中控制是为对象加上一个路径表达式,用以描述对象方法执行的各种并发限制.采用集中控制方式实现对象同步控制的不足之处在于:它要求设计者必须对对象的并发行为有整体的了解.另外,集中控制会带来继承异常[5],即在定义子类时,必须对父类的同步控制描述进行重定义,以把新定义的方法考虑进来.

分散控制方式是把对象的并发控制描述分布到对象的各方法上.其中一种是在对象各方法的实现中进行同步控制,对象无条件地接收消息,在消息的处理方法中,根据对象目前的状态决定是等待还是继续执行.这种控制方式的不足之处在于:被挂起的方法往往使得对象处于一种不稳定状态,使得对象此时不适合于处理其他消息,并且,当在子类中改变父类某方法的同步控制时,需对整个方法重定义.另一种分散控制方式是在对象类中给每个方法加一个激励条件,当对象接收到消息时,将根据相应方法的激励条件来决定是否处理该消息,对不满足激励条件的消息,则让其等待.这种控制方式的好处在于:当定义子类时,可以对父类中各方法的激励条件和方法的实现分开进行继承,从而避免了许多不必要的重定义.

本文采用如下的同步策略:在对象类中,为每个方法定义一个布尔表达式,该布尔表达式表示相应方法的激励条件.对象的缺省同步控制是顺序的条件同步,即在有多个满足并发限制(激励条件)的消息时,只允许其中一个消息的方法执行.这时,对象处于关闭状态,方法执行结束时自动打开对象,使得对象可以接受下一个满足同步条件的消息进行处理.为了实现对象内部的并发,一个方法在执行中也可以显式地开放对象,使得其他满足同步限制的方法可以得到执行,这样,一个对象中可以同时存在多个活动线程,对象的实现者应保证:一个方法在显式地开放对象后,不应再使用成员变量,以保证互斥.

在我们所采用的并发对象模型中,对象可以有一个体,一旦对象被创建,它的体作为单独的一个线程开始执行.这里,对象体的作用不是用于对象的并发控制,而是作为在对象模型中引进并发的一种机制.对象的并发控制代码由编

译器产生，对象接收消息时，自动调用这段代码，该代码将根据相应方法的激励条件和对象目前的状态(开放/关闭)来决定是否处理该消息.另外，当一个方法执行结束或在方法执行中显式地开放对象时，若有其他等待处理的消息，则对象的并发控制代码也会被调用.

2 在C++中引进并发机制

为了能用C++描述上述的并发面向对象模型，我们对其进行了扩充.首先设计了一个类:Concurrency，其定义如图1所示.所有并发对象必须是由Concurrency类的子类所创建.

```
class Concurrency
    { protected:
          void lock();
          void unlock();
          virtual void body();
          void run();
          void stop();
    }
```

图1 Concurrency类定义

在Concurrency类的定义中，(1) lock()用于关闭对象.当在其子类对象的某成员函数(方法)中调用了lock()，若对象处于开放状态，则关闭对象，该成员函数继续执行下去；否则，该成员函数等待其他成员函数开放(unlock)对象.成员函数重复调用lock()不使自己等待.(2) unlock()用于打开对象.当在其子类对象的某成员函数中调用了unlock()，若对象处于关闭状态，并且是由该成员函数调用lock()关闭的，则打开对象；否则，不做任何事情.(3) body()用于定义对象的体.Concurrency类中的body()为虚函数，它不做任何事情，子类中可以对其重定义.(4) run()用于创建一个线程以执行对象的体，若在子类中重新定义了body()，则创建子类对象时，执行的是子类的body().(5) stop()用于终止body()所在线程的执行.

若通过显式地调用Concurrency类中的成员函数来定义其他并发对象类，这将给程序设计者增加负担，并且也容易出错.因此，我们在C++类定义中增加了一个成分:concurrency_control，它用于给每个成员函数定义一个激励条件.这里，激励条件是一个C++布尔表达式，其中的运算分量可以是对象的成员变量和相应成员函数的形式参数.激励条件用于实现条件同步.图2给出了并发对象类BoundedBuffer的定义.

```
class BoundedBuffer: public Concurrency
{ concurrency_control:
    void put(char ch): (in+1)%max_len != out;
    char get(): in != out;
  protected:
    char buf[100];
    int in,out,max_len;
  public:
    BoundedBuffer()
    { in=out=0; max_len=100;
    };
    void put(char ch)
    { buf[in]=ch;
      in=(in+1)%max_len;
    };
    char get()
    { char ch1=buf[out];
      out=(out+1)%max_len;
      return ch1;
    }
}
```

图2 BoundedBuffer类的定义

在定义并发对象类时，其方法的激励条件可以从父类或祖先类继承.当在一个并发对象类中定义了一个方法，但

没有定义其激励条件时,如果该方法是对父类或祖先类中某方法的重定义,则该方法的激励条件继承父类或祖先类中相应方法的激励条件,否则,该方法没有激励条件,对象执行该方法的先决条件只要求对象处于开放状态.如果在一个并发对象类中定义了一个方法的激励条件,但没有定义该方法,则该激励条件是对父类或祖先类中相应方法激励条件的重定义.

一般来说,程序设计者不必在各成员函数中显式地调用 *lock*()和 *unlock*().因为,并发对象类的实现保证了对象的条件同步与互斥.为了允许对象内部的并发,在成员函数的执行中,可调用 *unlock*()释放对象,使得其他被锁住的消息得到处理.若显式地调用了 *unlock*(),则相应成员函数应保证在下一次调用 *lock*()前不对成员变量进行操作.在 BoundedBuffer 类中没有对 *body*()重新定义.若在某并发对象类中重新定义了 *body*(),则应在 *body*()的执行中适当的时候,调用 *unlock*()释放对象,以接收消息处理.

3　转换机制

为了使得扩充的并发对象类能够被 C++编译程序识别,我们设计了一个转换程序,其转换策略如下:对类中的每个成员函数 *m* 及其激励条件 *B*,先把 *m* 改名为_*m*,然后重新定义 *m*.图 3 给出了对 *m* 进行重定义的结果.若类或子类中的其他成员函数调用了 *m*(),则调用处改成_*m*(),即对象向自己发送消息不受对象并发控制的限制.

```
void m(...)
{ lock();
  while (! B) { unlock(); lock();}
  _m(...);
  unlock();
};
```

(a) 无返回值

```
type m(...)
{ lock();
  while (! B) { unlock(); lock(); }
  type result=_m(...);
  unlock();
  return result;
};
```

(b) 有返回值

图 3　成员函数的转换

当在类中没有给出成员函数 *m* 的激励条件时,若 *m* 不是对父类(祖先)的 *m* 重新定义,则在重新定义 *m* 时,没有 while (! B) { unlock(); lock(); }语句;否则,不重新定义 *m*,只把名改为_*m*,由父类的 *m* 处理条件同步.当在类中没有定义成员函数 *m*,而给出了 *m* 的激励条件(对父类成员函数 *m* 的激励条件重新定义)时,则只重新定义 *m*,而不产生_*m*,重新定义的 *m* 中将调用父类的_*m*.

为了能在创建对象结束时自动执行对象的 *body*(),转换时,对并发对象类的构造函数进行了处理.假设有并发对象类 *A* 和 *B*,*A* 的父类为 Concurrency,*B* 的父类为 *A*,现创建 *B* 类的对象 *b*.为了保证在 *B* 类的构造函数执行结束时调用 *run*(),而不是在其父类的构造函数中调用 *run*()(因为在执行父类的构造函数时,*B* 类的对象还没完全构造好),给 *A* 和 *B* 的每个构造函数增加一个缺省参数 *tag*,其缺省值为 *CREATE*.在 *B* 类构造函数的成员初始化符表中显式地给出对父类 *A* 构造函数的调用,调用时增加一个新参数 *NONCREATE*.若 *A* 和 *B* 中没有定义缺省构造函数,则给它们加上,并按上述规则给予改造;若在 *B* 的构造函数成员初始化符表中已经显式地调用了父类 *A* 的某构造函数,则只需给该调用加上 *NONCREATE* 参数.图 4 给出了对并发对象类构造函数的转换策略.

```
class A: public Concurrency
{ public:
    A(...,_TCreateTag  _tag=_CREATE)
    {...;
      if (_tag==_CREATE) run();
    }
}
class B: public A
{ public:
    B(...,_TCreateTag  _tag=_CREATE):A(...,_NONCREATE)
    {...;
      if (_tag==_CREATE) run();
    };
}
```

图 4　转换中对并发对象类构造函数的处理

为了在撤消对象时,终止 *body*()的执行,需在并发对象类的析构函数中调用 *stop*(). 图 5 给出了并发对象类 BoundedBuffer 的转换结果.图 6 给出了基于 Windows 95 所提供的多线程和同步设施而实现的 Concurrency 类.

```
class BoundedBuffer: public Concurrency
{ protected:
    ...;
    void_put(char ch) {...};
    char_get() {...};
  public:
    BoundedBuffer(_TCreateTag  _tag=_CREATE)
    {...
      if (_tag=      _CREATE) run();
    };
    ~BoundedBuffer()
    { stop();
    ...
    };
    void put(char ch)
    { lock();
      while (! ((in+1)%max_len ! =out))
      { unlock(); lock(); }
      _put(ch);
      unlock();
    };
    char get()
    { lock();
      while (! (in ! =out))
      { unlock(); lock(); }
      char result=_get();
      unlock();
      return result;
    }
}
```

图 5 BoundedBuffer 类定义的转换结果

```
#include (windows.h)
#include (afxmt.h)
class Concurrency
{ private:
    HANDLE mutex,thread;
    static DWORD new_thread(LPDWORD p)
    { ((Concurrency *)p)->body(); };
  protected:
    enum_TCreateTag {_CREATE,_NONCREATE};
    Concurrency()
    { mutex=CreateMutex(NULL,FALSE,NULL);
     thread=NULL;
    };
    ~Concurrency() { CloseHandle(mutex); };
    void lock();
    { WaitForSingleObject(mutex,INFINITE); };
    void unlock();
    { while (ReleaseMutex(mutex)); };
    void run()
    { DWORD id;
    thread=CreateThread(NULL,0,new_thread,this,0,&id);
    };
    void stop()
    { if (thread ! =NULL)
      { TerminateThread(thread,0);
        thread=NULL;
      }
    };
    virtual void body(){};
}
```

图 6 Concurrency 类的实现

并发对象类的使用与其他 C++类的使用完全相同,即先定义并发对象类的实例变量(对象),然后调用其成员函数对其操作.使用者不必考虑所使用的并发对象的并发控制问题,它由所使用的并发对象自己完成.另外,使用者也不必考虑如何创建并发对象体的执行线程,它由并发对象类创建对象时自动完成.因此,扩充的 C++既保留了原C++的风格,又具有描述并发执行的能力.

4 结 论

本文提出了一种对 C++进行并发扩充的方案,其核心在于引进了一个预定义的类 Concurrency 和在并发类中加入了描述条件同步的机制(激励条件).同时,还给出了把扩充的 C++类定义转换成 C++类定义的策略.如果不采用转换机制,程序设计者也可直接使用 Concurrency 类中所提供的 *lock*()与 *unlock*()等函数来实现对象的并发控制,当然,这给程序设计带来了麻烦.若采用转换方式来设计并发对象类,设计者一般不直接使用 Concurrency 类中所提供的设施来控制并发,它由转换程序来保证.当然,为了实现对象内部并发,使用者也可使用 *lock*()和 *unlock*(),但这只是为了解决一个对象各方法间对成员变量的互斥使用问题,复杂的条件同步控制已被方法的激励条件所吸收.

参考文献

1 Stroustrup B. The C++ Programming Language. Reading, Massachusetts: Addison-Wesley, 1986

2 Goldberg A, Robson D. Smalltalk-80: the Language and Its Implementation. Reading, Massachusetts: Addison-Wesley, 1983

3 Matuoka S, Yonezawa A. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In: Agha G *et al* eds. Research Directions in Concurrent Object-Oriented Programming. Cambridge, Massachusetts: MIT Press, 1993. 107~150

4 Agha G. Concurrent object-oriented programming. Communications of ACM, 1990,33(9):125~141

5 America P. POOL-T: a parallel object-oriented language. In: Yonezawa A, Tokoro M eds. Object-Oriented Concurrent

Programming. Cambridge, Massachusetts: MIT Press, 1987. 199～220
6 Meyer B. Systematic concurrent object-oriented programming. Communications of ACM, 1993,36(9):56～80
7 Papathomas M. Language design rationale and semantic framework for concurrent object-oriented programming [Ph. D. Dissertation]. University of Geneva, 1992
8 Yonezawa A *et al*. Modelling and programming in an object-oriented concurrent language ABCL/1. In: Yonezawa A, Tokoro M eds. Object-Oriented Concurrent Programming. Cambridge, Massachusetts: MIT Press, 1987. 55～89

An Approach to Concurrent Extension of C++

CHEN Jia-jun　ZHAO Jian-hua　ZHENG Guo-liang

(*Department of Computer Science and Technology　Nanjing University　Nanjing*　210093)
(*State Key Laboratory for Novel Software Technology　Nanjing University　Nanjing*　210093)

Abstract　In this paper, a concurrent object-oriented programming model is presented. It consists of a group of concurrent objects which may have bodies. When the concurrent object with a body is created, its body begin to run. In this model, the synchronous message passing is adopted for inter-object communication and the intra-object concurrency is permitted. The object concurrency control is distributed among each method's activation condition in an object. Based on the presented model, C++ is extended. A policy is also given to transform the classes defined in the extended C++ into C++ classes. The implementation uses the multithreading and synchronization mechanisms supported in some multitasking operating systems, such as Windows 95.

Key words　Object-oriented, concurrency, C++, concurrent extension, transf