计算机科学2001Vol. 28№. 5

并发对象中双层结构的同步控制模型

The Two-Layered Synchronization Control Model for Concurrent Object

王东奎　李国东　杨海荣　张德富

（南京大学计算机软件新技术国家重点实验室　南京210093）

Abstract　In order to solve "inheritance anomaly", we introduce a new kind of synchronization control model which divides the synchronization control into mutex control and condition synchronization control. Mutex control is completely implemented by pre-compling system and run-time system, so the developers merely concentrate on condition synchronization problem and don't consider mutex problem again when developing concurrent object-oriented programming, which decreases the occurrence of "inheritance anomaly" and reduces the difficulty of development to a great extend; Condition synchronization control adopts an approach to solving "inheritance anomaly" based on object's state.

Keywords　Concurrent object, Inheritance anomaly, Mutex control, Condition synchronization control

随着 Internet 网迅速普及，特别是对 NOW 研究的不断深入，以及并行计算机的推广应用，人们对并发程序的开发需求不断增加，而传统的功能分解方法给并发程序的复用带来了困难，因此有必要将面向对象方法融入并发程序设计以增强程序的易复用性和易扩充性。

继承和同步分别是面向对象程序设计和并发程序设计的主要特征，在并发面向对象程序设计中，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约束代码。然而，当同步约束条件与方法定义融合在一起时，就可能对父类中原本可以继承的方法进行重定义，从而产生一系列的问题，即继承异常[1]。

人们在试图解决继承异常问题时，提出了很多同步控制策略，一是集中控制方式：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，A）用对象体来控制对象何时接收消息[2]；B）为对象加上一个路径表达式[3-4]，用以描述对象方法执行的各种并发限制。二是分散控制方式：把对象的并发控制描述分布到对象的各方法上，A）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；B）在对象类中给每个方法加一个激励条件[5]，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消息，对不满足激励条件的消息，则让其等待。

通过分析，我们发现：以上的同步控制策略没有明确地区分互斥和条件同步控制。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，如对于有界缓存（Bounded Buffer）对象，当缓存不空不满时，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），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。

对象的条件同步是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。

传统的同步控制策略将互斥和条件同步交织在一起，为了使开发人员从烦琐的互斥控制中解脱出来，把互斥控制完全交给预编译和运行系统去处理，这正是本文的目的所在。

1　互斥控制

传统的互斥控制是由开发人员编写互斥控制代码来解决一个对象内各方法间对成员变量的互斥访问，同时并发程序设计语言必须提供诸如信号量（semaphore）、临界区（critical region）、管程（monitor）等设施。开发人员必须知道方法在哪些地方访问了成员变量，如何控制对这些成员变量的互斥访问，当增加或修

王东奎　硕士生，主要研究领域为并行/分布处理技术、面向对象语言。李国东　硕士生，主要研究领域为分布对象技术。杨海荣　硕士生，主要研究领域为并行/分布处理技术。张德富　教授，博士生导师，主要研究领域为并行处理技术和分布式系统。

改用户类中的方法时，相应的互斥控制代码也要相应地改变，降低了代码的可复用性和可扩充性，给开发人员带来极大的不便，大大增加了继承异常发生的概率。本文采纳了系统自动实现对象成员变量的互斥访问：把对成员变量的互斥访问交给预编译系统和运行系统去实现。

1.1 实现策略

(1)预编译器为每个类生成一张读写表(如表1，具体实现见1.2小节读写信息的获取)，每个表项对应着类中某个方法的读写集。

表1

方法名 Method	读集 W_{method}	写集 W_{method}

(2)运行时每个对象都有一个同步控制线程(对象创建时系统为每个对象创建并启动一个同步控制线程，该线程在对象的生存周期内监听着发往该对象的方法请求)，当有方法请求到来时，同步控制器在方法请求队列(表2)中插入新的一项，表项中的"有依赖关系(依赖关系的判断见1.3小节)的前驱方法集合 Pre"是根据对象相应类的读写表信息生成的。

表2

编号i	方法名 Method	有依赖关系的前驱方法集合 Pre	运行标志 Run

其中 Pre 是编号 j 组成的集合，j(j<i)是跟方法 method 存在依赖关系的某个前驱方法在队列中的编号；Run 表明该方法是否已经分配运行，值为 True 和 False。

(3)同步控制器就根据方法请求队列来实施方法的互斥调用。如果方法请求队列中某方法的 Pre 集合为空，则说明该方法跟对象中的其它方法没有访问冲突，可以立即投入运行；如果 Pre 非空，则在队列中等待。

1.2 读写信息的获取

考虑到 Java[6]的语言定义比较清晰和标准，下面的方法都以 Java 的类定义和实现为样板。对于 Java 来说，类所涉及的读写集 S 由三部分组成：它所继承的父类的域(Field)成员，它所继承的所有接口(Interface)的域成员和它自身定义的域成员。所继承的域成员不包含在父类或接口中对它不可见的部分。

可供客户调用的对象方法集 M 由它的类所定义的 public 方法和继承的 public 方法组成。预编译器为每一个方法 p 生成两个集合 R_p，W_p。我们用这两个集合表示每一个方法的读写信息。

Java 表达式中共有12种赋值操作符，对于方法 p，如果 v 出现在操作的左边，其中 v∈S，则 $W_p=W_p\cup\{v\}$。而对 v 在 p 中的其他出现则有 $R_p=R_p\cup\{v\}$。这里有一种 v 存在别名的情况，例如下面的代码：

```
SOMECLASS obj=new SOMECLASS();
SOMECLASS another_obj=obj;
```

赋值以后对 another_obj 的操作事实上也作用在 obj 上，another_obj 被称为 obj 的别名。预编译程序扫描方法 p 的时候先建立原名和别名的对应关系，然后在求方法概要信息时将别名转换为原名加入相应的读写集。

方法中 p 可能出现函数调用 foo(arg1,arg2,…,argn)，这时读写集的求法遵循以下步骤：1)如果 argi∈S，则 $R_p=R_p\cup\{argi\}$，以引用方式传递参数 argv，如果 argv∈S，将函数 foo 中相应的变量列为 argv 的别名，继续分析函数内的代码，完成后将 argv 的这个别名删除。2)如果 foo=p，或者 foo 中含有对 p 的调用，也就是递归的情形，只将 p 作为一个基本表达式看待，然后继续分析调用点以下的代码。3)如果 foo≠p，则继续分析 foo 的代码。

下面我们以矩形类 Rect 为例来说明读写信息的获取：

```
class Rect{
  int x,y; //左上角坐标
  int width,height; //矩形的宽和高
  Rect(){ //构造方法
      x=0;y=0;width=300;height=200;
  }
  void move (int x,int y){
        //将矩形的左上角移动到(x,y)处
      this.x=x;
      this.y=y;
  }
  boolean inside (int x,int y){
        //判断点(x,y)是否处于矩形内部
      return (x>=this.x) && ((x-this.x)<this.width)
                  &&
             (y>=this.y) && ((y-this.y)<this.height);
  }
}
```

预编译器为 Rect 类生成读写表如表3。有两点要注意一下：方法的读写集不牵涉到方法的局部变量，这些局部变量对对象内的其它方法来讲是不可访问的，也就不存在共享访问带来的互斥控制问题；另外也不必为对象的构造方法生成读写信息，因为对象的其它方法请求必须在对象创建后才能产生，此时构造方法肯定已调用完毕且只调用一次，不会带来访问冲突的问题。

表3

方法名 Method	读集 R_{method}	写集 W_{method}
Move	Φ	{x,y}
Inside	{x,y,width,height}	Φ

1.3 依赖关系的判断

假设 p、q 为并发对象类定义中的两个方法，R_p、R_q 为 p、q 执行过程中读取的父类、所有父接口和这个接口类自身定义的变量的集合，W_p、W_q 为 p、q 执行过程中修改的父类、所有父接口和这个接口类自身定义的变量的集合。通常，如果 $(R_p \cap W_q) \cup (W_p \cap R_q) \cup (W_p \cap W_q) \neq \Phi$，则方法 p、q 间存在依赖关系，否则没有依赖关系。

2 条件同步控制

通过互斥控制后，接着就进行条件同步控制检查，为了减少继承异常的发生，我们采纳了将条件同步代码跟方法体分离的策略，这样条件同步控制实际上就分成两个子部分：一是前置条件判断，根据状态表(见表4)判定当前的方法调用是否可执行，若可执行，则创建线程执行方法，否则在方法请求队列中等待；二是后置状态转换，即具体方法执行结束后对状态表的修改。

2.1 实现策略

1)创建对象时，运行系统根据类的状态定义为每个对象建立状态表。

2)同步控制器根据状态表，判断在当前状态下方法是否可执行，若可执行，则创建线程运行该方法，否则在方法请求队列中等待。

3)方法运行结束后，根据类定义中的状态转换修改相应的状态。

2.2 状态定义(类定义中的 state 部分)

状态采用表达式来定义：state_name = expression，其中 expression 是用状态控制量来定义的。

2.3 状态与可接受方法集合的对应问题(类定义中的 relation 部分)

采用方法集合来定义状态的可接受方法：M(state_name) = method_set，通过建立状态 state_name 与方法集合 method_set 的对应关系来表示当对象处于状态 state_name 时，可以接收的方法。

2.4 状态转换(类定义中的 transmition 部分)

通过定义一个状态转换部分 method.set⇒expressions 来完成，method.set 中的方法执行结束后还需完成的动作。

2.5 实例

下面，我们就以先入先出(FIFO)的有界缓冲区为例来说明条件同步控制。有界缓冲区类有两个公共方法 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。

表4 状态表

状态集	满足条件	可执行方法
状态1	条件1	方法 i,…
状态2	条件2	方法 j,…
…	…	…
状态 n	条件 n	方法 k,…

表5 实例

状态集	满足条件	可执行方法
EMPTY	size=0	put
FULL	size=MAX	get
PART	0<size<MAX	put,get

先入先出(FIFO)的有界缓冲区类定义如下：

```
class Buffer{
    int size,buf [MAX],…;//状态控制量
    State: EMPTY:=(size=0);//状态定义
           PARTIAL:=(0<size<MAX);
           FULL:=(size==MAX);
    relation. M(EMPTY)={put};//某状态下的可执行方法定义
              M(PARTIAL)={get,put};
              M(FULL)={get}
    public
          int get(){…} //取走一元素并修改状态控制量
          void put(){…} //插入一元素并修改状态控制量
    transmition: {get}⇒{size--;}
                 {put}⇒{size++;}
}
```

根据 Buffer 类的定义可以形成如表5所示的相应表。同步控制器就根据状态表来实施条件同步控制。

3 双层结构的同步控制模型

3.1 并发对象模型

本文提出的并发对象模型(图1)中，所有对象都为并发对象，多个并发对象可以同时运行，同一个对象内亦可有多个并发线程，从而支持对象间和对象内两级并行。

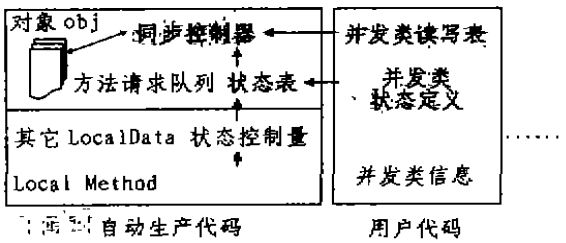

图1 并发对象的结构

对象间的消息传递都采用异步消息发送方式和需要时等待[9]的策略，即消息发送对象不必等待消息返回即可继续执行，当需要访问此消息结果时，系统自动检测结果是否已返回。如果尚未返回，则执行线程阻塞。

系统为每个对象创建一个控制线程，对象被创建时就启动控制线程，控制线程根据方法请求队列、状态表、读写表来实施对对象方法的互斥和条件同步控制。

3.2 同步控制模型

本文提出的同步控制模型是基于分层结构的同步控制模型，由互斥控制和条件同步控制构成。任何方法从请求的产生到执行必须经过互斥控制和条件同步控制，只有通过互斥和条件同步控制，方法才能投入运行，否则被挂起直到同步控制器的再次调用。

同步控制器实际上是一个控制线程(具体流程见图2)，在对象创建时系统为每个对象创建并启动一个同步控制线程，该线程在对象的生存周期内监听着两类事件：新的方法请求和方法运行结束事件。新的方法请求到来时，需对之进行互斥控制和条件同步控制，只有两者皆满足时方可创建线程运行方法；方法运行结束时，需要进行善后处理，如修改方法请求队列中相关方法的 Pre 集合和状态，前者导致了方法请求队列中各方法间依赖关系的改变，后者使得原来被条件同步控制阻塞的方法又有可能满足条件同步控制，这样需要对方法请求队列中的方法进行重新调度，将满足互斥控制和条件同步控制的方法投入运行。

其中 Length(queue)：获取请求队列的长度，也就是等待和正在执行的方法个数。这里并没有从队列中丢掉正在执行的方法，是因为未执行完的方法尚未完成对读写集的更新，因而下面的方法执行还有可能依赖于它的结果。

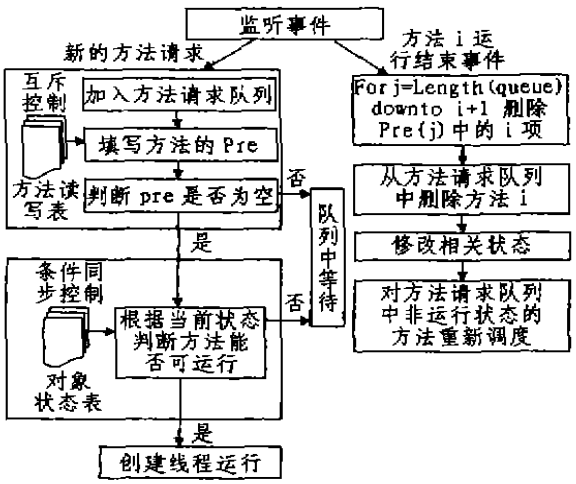

图2 同步控制模型流程

4 实例

本文提出的同步控制模型的关键是为了解决互斥问题，而条件同步采用了比较传统的基于对象状态的继承异常解决方案，所以我们这里的实例主要针对本文提出的互斥控制策略，下面我们以"读写者"互斥控制为例来说明本文的互斥控制策略。"读写者"的类定义如下：

```
public class ReadWrite
{public int a,
ReadWrite(){a=0;}
public void Read(){println(a);}
public void Write(){a=a+10;}
}
```

(1)预编译器为 ReadWrite 类生成一张读写表(见表6)。

表6

方法名 Method	读集 R_{method}	写集 W_{method}
Read	{a}	Φ
Write	{a}	{a}

(2)运行时刻创建 ReadWrite 类对象 objRW，同时创建并启动一个控制线程，控制线程根据方法请求队列、状态表、读写表来实施对对象方法的互斥和条件同步控制。

假设向 objRW 发出的方法请求顺序为 Read1，Read2，Write3，Read4(这里的序号只是表示消息到达的先后，Read1，Read2，Read4调用相同方法 Read 的代码)。

(3)方法请求 Read1到达时，同步控制器先把 Read1加入方法请求队列，由于方法 Read1没有前驱方法，所以 Pre 为空，顺利通过互斥(读写者仅涉及互斥控制)检查，并投入运行。

(4)方法请求 Read2到达时，同步控制器先把 Read1加入方法请求队列，由于跟前驱方法 Read1没有依赖关系，所以 Pre 为空，通过互斥检查，并投入运行。

(5)方法请求 Write3到达时，同步控制器先把 Write3加入方法请求队列，由于跟前驱方法 Read1、Read2有依赖关系，被互斥检查阻塞，只能在方法请求队列中等待。

(6)方法请求 Read4到达时，同步控制器先把 Read4加入方法请求队列，由于跟前驱方法 Write3有依赖关系，被互斥检查阻塞，也只能在方法请求队列中等待。

结果就形成表7所示的方法请求队列，方法请求 Read1、Read2并发运行，实现了多个读者可以同时进行读操作，从而实现了对象内的并发性。

(7)方法 Read1运行结束，同步控制器监听到这个事件，修改方法请求队列中的 Pre 集合，将跟方法 Read1有关的项从集合中删除，并将该方法从方法请求队列中删除，同步控制器对方法请求队列中的方法重新调度，Read2满足同步控制并投入运行，结果形成表8。

表7

编号 I	method	Pre	Run
1	Read	Φ	True
2	Read	Φ	True
3	Write	{1,2}	False
4	Read	{3}	False

表8

编号 I	Method	Pre	Run
2	Read	Φ	True
3	Write	{2}	False
4	Read	{3}	False

(8)方法 Read2运行结束后，Write3投入运行，方法请求队列变为表9。

(9)方法 Write3运行结束后，Read4投入运行，方法请求队列变为表10。

表9

编号 I	method	Pre	Run
3	Write	Φ	True
4	Read	{3}	False

表10

编号 I	Method	Pre	Run
4	Read	Φ	True

由于达到 obiRW 对象的方法请求是动态的，在任何时刻都可能有新的方法请求到来，读者自己可以试验一下新的方法请求到来时对方法请求队列的影响。

小结 本文提出的同步控制模型包括互斥控制与条件同步控制。为自动实现互斥控制部分，本文扩充了预编译系统和运行系统，不仅大大减少了继承异常的发生，而且降低了开发人员进行并发面向对象程序设计的难度。条件同步控制采纳了基于对象状态的继承异常解决方案，实现控制代码跟方法体的分离，解决了部分代码的复用、进一步降低了继承异常的发生。

随着 Internet 网迅速普及，特别是对 NOW 研究不断深入，我们可以把同步控制扩展到 NOW 环境，将通过同步控制的方法请求分布到另外的空闲工作站上运行，以提高对象内的并行性。另外可以改进依赖关系的判定算法，剔除一些假依赖关系，进一步提高对象内的并行性。

参 考 文 献

1 Matsuoka S, Yonezawa A. Analysis of inheritance anomaly in object-oriented concurrent language. In: Agha G, Wegner P, Yonezawa A, eds. Research Directions in Concurrent Object-Oriented Programming. Cambridge, Mass: MIT press, 1993. 107～150

2 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

3 Andler S. Predicate path expressions. In: Proceedings of 6 POPL, ACM, 1979

4 Ben-Arm. Principle of concurrent and distributed programming. Prentice-Hal International, Hemel Hempstead, 1991

5 陈家俊，赵建华，郑国梁. C++的一种并发扩充方案. 软件学报，1998，8：586～591

6 Cornell G, Horstmann C S. Core Java. SunSoft Press, 1996

7 Yonezawa A, et al. Modelling and programming in an object-oriented concurrent language ABCL/1. In: Yonezawa A, Todoro M, eds. Object-Oriented Concurrent Programming. Cambridge, Massachusetts: MIT Press, 1987. 55～89

(上接第33页)

4.3 面向对象链接信息系统的困难

面向对象链接信息系统的难点集中体现在两个方面：节点结构化的规模及其技术手段。节点结构化规模过小，意味着系统检索入口少、动态聚合只能在少数几个维度展开；反之规模过大，又意味着标引繁复、甚至会出现高阶信息数据量远大于0阶信息数据量的情形。节点结构化有两种可能的技术手段：人工标引或机器标引。人工标引可能会受到作者习惯性惰性的抵制；机器标引又将遭遇全文智能检索软件所面临的技术困难。机器标引还会带来另一个问题：既然创作与标引分离，那么是否有必要保持0阶信息与高阶信息一体化的节点结构呢？我们是否只需将管理 WWW 这一浩瀚信息资源库的期望寄托在搜索引擎智能的全面提升之上呢？

这些问题的实质，仍然是信息管理成本与效益之间的矛盾冲突。成本与效益折衷会倾向于什么样的信息管理结构呢？我们无法断言，只能等待未来技术与商业的发展作出抉择。

参 考 文 献

1 Tim Berners-Lee Information Management: A Proposal 1989.3, 1990.5. Available at: http://www.w3c.org.

2 李幼平. 构思"第五传媒". 北京·计算机世界周报，vol.49-D1，1998.12.21

3 高杨，李幼平. 内容主动服务技术. 北京·计算机世界周报，vol.50-D7，1998.12.28

4 王云，赵武文，高杨. 内容表示语言与统一内容定位. 北京·计算机世界周报，vol.50-D9，1998 12.28

5 唐世谓，杨冬青，王军. 互连信息空间 InterSpace. 北京：计算机世界周报，C5版，2000.2.21