面向对象程序设计模型中的并发行为

摘 要

面向对象软件开发技术已广泛应用于软件系统的设计与构造，面向对象模型是对客观世界活动的自然刻划，其中的对象是对客观世界中有形或无形实体的直接模拟。用面向对象技术开发软件，能够减小问题域与解题域之间的语义间隙，使得软件开发过程显得比较自然，从而可提高软件生产率。

客观世界中的活动往往是并发进行的，而目前大多数面向对象模型只提供了描述系统顺序执行的能力。为了加强面向对象模型的表达能力，必须在面向对象模型中提供并发描述机制，使其能够描述客观世界中的并发行为。

本论文工作的主要目的就是研究如何在面向对象模型中引进并发描述机制，使其能够描述客观活动的并发行为，并且，所引进的并发描述机制要与面向对象模型相容，不破坏面向对象模型中已有的一些特性。

本论文首先分析了在面向对象模型中引进并发所面临的各种问题，并给出了现有的一些解决方案。然后，在研究了现有的一些并发面向对象模型的基础上，论文提出了一个基于并发对象的并发面向对象模型，在该模型中，并发机制以一种与面向对象模型相集成的方式被引进。对象可以有私有执行线程，各对象的私有线程之间以及对象内部的各线程之间并发执行；对象间采用远程过程调用方式进行通信，对象对消息的接收以及内部各线程之间的并发执行进行控制；并发对象类的并发属性可以从父类继承。论文还用CSP对所提模型中并发对象的语义进行了形式化描述，论文最后用所提模型对C++进行了并发扩充，使得C++能够用于描述并发计算。

关键词：面向对象，程序设计，并发对象，CSP，C++

Concurrent Behavior in Object-Oriented Programming Models

Abstract

Object-oriented(OO) technique has been widely used in the design and construction of software systems. Object-oriented models describe the real world activities in a natural way. The objects in the models are the direct simulations of the entities in the real or mind world. Developing software with OO technique can minimize the semantic gap between problems and solutions. It makes software development processes more natural and, thus, promotes software productivity.

The real-world activities are usually carried out concurrently. Currently, OO models only support the specification of the sequential behaviors of a system. To model the concurrent behaviors of the real-world activities, concurrency must be introduced into OO models.

The main concern of this dissertation is to study how to introduce concurrency into OO models. The introduced concurrent mechanism should be compatible with OO models and does not hamper the existing properties of OO models.

In this dissertation, all kinds of problems encountered in introducing concurrency into OO models are analyzed first. Then, based on the study of some existing concurrent object-oriented models, the dissertation present a concurrent object-oriented model around concurrent objects. In the model, concurrency is introduced in an integrated way. Objects may have private threads. Threads in objects execute concurrently. The remote procedure calls are used as communication mechanism between objects. Objects control their own message accepting and the concurrent executions of their private threads. The specification of concurrency control in a class can be inherited by its subclasses. The formal semantics of the concurrent objects in the presented model has been given in CSP. At last, C++ has been extended based on the presented concurrent object-oriented model.

Keywords: Object-Oriented, Programming, Concurrent Objects, CSP, C++

目 录

第一章 前言

面向对象软件开发技术已广泛应用于软件系统的设计与构造。面向对象模型由一组对象构成，对象之间通过发送消息进行交互，这种模型是对客观世界活动的自然刻划，其中的对象是对客观世界中有形或无形实体的直接模拟。用面向对象技术开发软件，能够减小问题域与解题域之间的语义间隙，使得软件开发过程显得比较自然，从而可提高软件生产率。面向对象技术中的模块、封装、数据抽象及继承等特性，使得它更适合于构造软件系统，并且构造出的系统具有较好的易维护性与可重用性。

客观世界中的活动往往是并发进行的，而目前大多数面向对象模型只提供了描述系统顺序执行的能力。为了加强面向对象模型的表达能力，必须在面向对象模型中提供并发描述机制，使其能够描述客观世界中的并发行为。在面向对象模型中，各对象间通过发送消息，相互影响，协作完成系统功能，其中存在着潜在的并发执行能力。为了保证面向对象并发系统的正确执行，必须把这种潜在的并发行为显式地表示出来，并且，所采用的并发描述机制应与面向对象模型相一致，不损害面向对象模型已有的一些重要特性，如：封装与继承等。

本论文工作的主要目的就是研究如何在面向对象模型中引进并发描述机制，使其能够描述客观活动的并发行为，并且，所引进的并发描述机制要与面向对象模型相容，不破坏面向对象模型中已有的一些特性。在本章中，首先对面向对象模型中的一些主要概念进行简要介绍；然后给出了对面向对象模型进行并发扩充应遵循的基本原则；最后介绍本论文的主要工作和论文的安排。

1.1 面向对象模型

面向对象技术首先成熟于面向对象程序设计(OOP)，然后逐步应用于软件的设计和软件的需求分析，产生了面向对象设计(OOD)和面向对象分析(OOA)[12,23]。用面向对象技术开发的系统由一组交互的软件对象构成，这些软件对象往往对应着实际应用中一些有形或无形的实体。面向对象模型包含了许多良好的程序设计思想，如：模块、封装、数据抽象、多态和继承等等，这些对软件的构造、维护及重用是很有用的。下面将对面向对象模型中的一些基本概念作简单描述。

1.1.1 对象与消息

面向对象模型由一组互相发送消息的对象构成，其核心概念就是对象和消息。对象是一个可标识的实体，它由状态和操作两部分构成，对象的状态由对象的局部变量(实例变量)来表示，对象状态的改变是通过向其发送消息进行的。当对象接收到消息后，将调用其某个方法(成员函数)来对该消息进行处理，方法的执行中有可能改变对象局部变量的值(对象的状态)。对象的方法构成了对象的接口，获取或改变对象的状态只能通过接口中的方法来进行。对象的这种封装特性可以看作是对模块化和数据抽象的支持。

1.1.2 类与继承

类刻划了一组具有共同特性的对象，在面向对象程序设计语言中，类是创建对象(类的实例)的模板(Template)。

类继承是一种资源共享机制，它使得在定义新类时可利用已有类的一些信息，这些已有的类被称为父类或超类，新定义的类被称为子类或衍生类。如只允许从一个类继承，则称为单继承，否则，称为多继承。类继承是支持软件复用的一个重要语言机制。

一个程序设计语言被称为面向对象的(Object-Oriented)，它首先要支持对象概念，除此之外，它还应支持类和继承。仅支持对象的语言只能被称为基于对象的(Object-Based)[89]，如：Ada[13]。

1.1.3 多态和动态定连

多态是指某一论域中的元素有多种解释。在面向对象程序设计语言中，多态主要是指一个父类指引元变量在程序运行时可指向父类和子类对象。

定连是指结构成分与其有关性质的确定和关联。在编译时刻完成的定连称为静态定连，否则，称为动态定连。在面向对象程序设计语言中，由于多态指引元变量的存在，往往需要动态定连。

1.1.4 类型与子类型

与类和继承相关的概念是类型和子类型。在很多面向对象程序设计语言中，用类表示对象类型，用类继承表示子类型关系，如：C++[82]、Eiffel[63]等，这样做的好处在于简化程序设计，并且使静态类型检查变得容易，但同时也带来一些问题，它造成语言的描述能力和程序的可靠性下降。因为，类实际上是给出了抽象数据类型的实现，而类继承则是类之间的一种代码复用机制，若用类表示类型则限制了类型的实现，用类继承表示子类型关系则要求类型间的子类型关系必须基

于类之间的继承关系，即使两个类的抽象行为存在子类型关系，如果它们没有继承关系，则编译系统不会认为它们是子类型关系。另外，因为在子类中可以对父类中的方法重定义，这样就有可能使得子类与父类有不同的行为，从而破坏子类型关系。因此，在一些面向对象语言中把类与类型分开考虑[8,53]。

1.1.5 相关的术语

在有关面向对象技术的不同研究中，往往对一个相同的概念采用了不同的术语描述，为了在论文的描述中不至于产生混淆，现将这些术语进行归类，同一类中的术语表示相同的概念，本论文中经常交替使用它们。

(1)对象、类的实例。

(2)对象的局部变量、成员变量、实例变量。

(3)父类、超类。

(4)子类、衍生类。

(5)方法、成员函数、操作、消息处理过程。

1.2 并发与面向对象模型

并发(concurrency)的概念伴随我们已有很长时间，几乎自从计算机出现起，并发就受到了人们的重视，它反映了程序潜在的并行执行能力。

传统的并发模型是围绕进程的概念进行设计的，它由一组进程构成，每个进程是一个顺序执行的过程，各进程间可以并发执行。进程在执行中可以相互通信，交换数据。进程间通信可以采用两种方式：共享变量和消息传递，信号量、临界区、管程和路径表达式等被用来对并发进程的操作进行同步。我们认为，进程是一个实现级的概念，它是对客观世界活动的一种间接模拟，因此，采用进程模型来解决客观世界中的并发问题就显得极不自然，并且也使得并发程序难以设计和理解。

面向对象模型以一种更加直接的方式刻划客观世界中的活动，模型中存在着潜在的并发执行能力。通过简单的观察，我们不难看出：一个对象向另一个对象发送消息后，若不需要或不立即需要消息的处理结果，前者不必等待后者处理消息，消息发送者和消息接受者可以并发执行；对象不都是处于被动的提供服务状态，它们中的一些除了能通过接收消息向外提供服务外，还可以有自己的事务处理；一个对象往往可以同时处理多个消息，等等。

为了保证面向对象系统的正确运行，必须要给出显式的并发描述与控制，否则将会出现混乱，例如：当处于某状态下的一个对象不能处理某消息时，强行调用该消息的处理方法将造成对象内部状态的破坏。另外，对一个对象的多个方法进行并发调用，也可能造成该对象内部状态的不完整和不一致。

如何把并发与面向对象相结合？如何在面向对象模型引进并发描述机制？ 目前很多人都在进行这方面的研究[3,10,11,19,64,71,72,85,91,92,7,56]，这些研究的做法各不相同，归结起来可分为两条途径：

(a)在面向对象模型中引进并发机制。
(b)在传统并发模型中引进面向对象思想。

不同的观点将产生不同的结果。(a)充分利用面向对象技术刻划客观世界的良好模型能力和面向对象的各个重要特性，同时把其潜在的并发能力显式地描述出来，使其适合于描述并发计算。(b)只是采用面向对象技术的某些思想，不是完全的面向对象，往往舍弃了面向对象的一些重要特性，从而削弱了面向对象的模型能力，采用(b)观点的研究结果一般是基于对象(Object-Based)而不是面向对象(Object-Oriented)的并发系统。因此，采用(a)是一个可取的途径，它从对象模型本身出发研究其并发能力，这样，既保留了面向对象的各个重要特性，又解决并发程序设计问题。

然而，很多的研究表明：并发与面向对象不是正交的，并发往往与面向对象的一些特性相冲突[59,62,85,49,71]。即使采用途径(a)来解决并发与面向对象结合问题，也存在不同的做法，这些做法对面向对象的支持程度是不同的。例如，在一些并发面向对象模型中就舍弃了继承特性[10]。因此，在面向对象模型中引进并发机制是一个方面，另一方面是要保证所引进的并发机制与面向对象模型相容，不破坏面向对象模型已有的一些重要特性。

1.3 主要工作

本文首先给出了在面向对象模型中引进并发应考虑的各种问题，然后，在对这些问题及现有的一些并发面向对象模型进行分析的基础上，提出了一种基于并发对象的并发面向对象模型，该模型主要考虑的是：

(1)对象的自治性。

(2)对象的并发控制与对象的功能分开进行描述。

(3)对象内部存在并发。

(4)对象间采用同步消息发送。

(5)对象可以有自己的执行线程。

(6)模型概念简单，不给程序设计者带来太大的负担。

论文还对所提并发面向对象模型中并发对象的语义进行了描述，最后，给出了采用论文所提模型对C++进行并发扩充的一种方案。

1.4 论文安排

本论文的安排如下：

第二章分析了在面向对象模型中引进并发描述所面临的问题以及目前的一些解决方案，并介绍了现有的一些并发面向对象模型。

第三章提出了一种基于并发对象的并发面向对象模型。

第四章对所提模型中并发对象的语义给出形式化描述。

第五章给出了用论文所提模型对C++进行扩充的一种方案。

第六章给出论文工作的总结及进一步的工作。

第二章 在面向对象模型中引进并发

顺序面向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它对象发送消息，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，即消息发送者调用消息接受者的某个方法，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，因此，整个系统只有一个执行线程(thread)。

如何在上述的顺序面向对象模型中引进并发行为？如何实现对象的并发控制？这是很多人都在研究的问题[3,10,11,19,64,71,72,85,91,92,7,56]。本章首先对传统的并发系统模型作一简单描述，然后给出在面向对象模型中引入并发行为应解决的问题和一些解决方案，最后介绍现有的一些并发面向对象模型。

2.1 并发系统的一般模型

人们习惯于编写顺序程序，理由是：顺序程序的行为完全在程序设计者的控制之下，但是，把一个程序构造成由若干个并发执行的部分组成往往会显得更加自然，并且，这样的程序能够充分发挥一些并行体系结构的计算机系统所提供的强大计算能力。虽然从理论上讲，一个设计良好的编译器可以发现一个顺序程序中潜在的并行性，并产生相应的可执行并发代码，但实际上，这只对一些有限的应用可行，对于一般的应用来说，为了描述问题领域中的并发活动，程序设计者必须在程序中给出显式的并发描述。

在一个并发系统中，计算任务是由若干独立(异步)执行的活动相互合作完成的。为了显式地描述这样的系统，一般要考虑三个方面的问题[6]：

(1)并发执行。

描述一个并发系统，首先要考虑采用什么样的结构来表示并发执行的单位，以及如何让它们并发执行。在传统的基于进程的并发系统中，描述并发结构的方式有多种，如：coroutine[4]、fork/join[25,28]、cobegin/coend[29]以及task[13]等，这些并发结构在抽象级别及表达能力上各不相同，其中，有些是属于低级描述但具有较强的表达能力，如fork/join；有些抽象级别较高但表达能力受到限制，如：cobegin/coend。另外，这些并发结构在描述进程的创建方面也各不相同，在有些结

构中，进程是静态创建的，进程的个数是固定的；在另一些结构中，进程的创建则是动态的，进程的个数随着系统的执行在变化。

(2)并发活动之间的通信。

在一个并发系统中，并发执行的活动之间往往需要进行交互，即，通信。在基于进程的并发系统中，进程间的通信可以采用共享变量(shared variables)和消息传递(message passing)两种方式，共享变量通信方式适合于具有共享内存体系结构的计算机中，消息传递则不受共享内存的限制。在消息传递通信方式中，还存在两种通信形式：异步消息传递(asynchronous message passing)和同步消息传递(synchronous message passing)，在异步消息传递中，消息发送者不必等待消息接受者接收消息，而是把消息放入一个消息缓存中，消息发送者即可继续执行下去，消息接受者适时地从消息缓存中接收消息进行处理，消息处理结果也以同样方式送给消息发送者。在同步消息传递中，消息发送者必须等到消息接受者接收消息后才能继续执行下去。

异步消息传递特点在于：它使得系统有较高的并行度，不足之处是：使得系统的行为难以理解，当消息接受者接收到某个消息时，消息发送者的状态可能已经不是发送该消息时的状态了。另外，采用异步消息传递，消息发送者还要考虑同步问题，即何时及如何获得回答消息。同步消息传递的特点是：消息发送就隐含了同步，不足的是使得系统的并发度降低，并且，如果处理不当易造成死锁。

异步消息传递和同步消息传递都是单向通信，为了方便地描述client/server计算模型中的消息传递，一种较高层次的消息传递机制被引进：远程过程调用(remote procedure call或RPC)。采用RPC方式进行消息传递时，消息发送者通过执行一个call语句来发送消息，从形式上看，call语句类似于通常的过程调用。当call语句执行时，首先把参数传给消息接受者，然后等待消息接受者执行相应操作并返回处理结果。远程过程调用属于一种双向通信。

(3)并发活动的同步。

在并发系统中，进行通信的并发活动间往往需要同步。较常见的同步形式是条件同步(condition synchronization)，即，一个活动的继续执行必须要等待一个条件或事件的发生。在基于共享变量的消息传递机制中还存在另一种形式的同步：互

斥(mutual exclusion)，即，为了保证用于通信的共享变量不受错误的并发操作的影响，往往要互斥地使用它们。

在传统的并发模型中，实现同步的机制有多种，其中，基于共享变量的同步机制有：信号量(semaphores[29])、条件临界区(conditional critical regions[42])、管程(monitors[29])和路径表达式(path expressions[18])等。信号量具有很强的表达能力，它几乎可以描述各种同步问题，但它属于一种无结构的低级设施，使用不当(如：漏掉一个P或V，或P和V颠倒了)将会产生灾难性结果；条件临界区相对来说更有结构一些，但与信号量一样，同步控制代码分散在各个进程中，如要了解某个共享资源的使用情况，必须要阅读整个程序；管程实现了共享资源的集中管理，它封装了共享资源及可施于其上的操作，管程的实现保证了对其操作的互斥调用，但是，管程在实现条件同步方面就显得不是那么有结构，必须借助于其它设施来完成，如：条件变量上的wait和signal操作，另外，嵌套的管程调用也会引起很多问题；路径表达式在描述操作间的并发限制上提供了比较优雅的方案，对描述并发计算的语义起到了积极的作用。路径表达式比较适合于描述与操作历史有关的同步问题，但在描述条件同步方面却显得无所适从，因为，一个操作是否可以执行往往要依赖于所操作的资源的状态，而有些状态并不直接反映在操作历史中。

当用消息传递进行通信时，消息传递就隐含了一种同步，即，消息只有在发送后才能收到，它给出了消息发送者与接收者之间的一种执行次序。消息传递机制如果设计得好，它就能既实现进程间的通信，又能实现进程间的同步。

2.2 面向对象模型的并发执行

进程是传统并发程序设计模型中的基本构件，同时它也是引起系统并发的基本单位。而在面向对象软件系统中，对象是其基本构件，同时，对象也是产生并发的自然单位，用面向对象模型来解决客观世界中的并发问题会显得更加自然。要在对象模型中引进并发，首先要考虑如何让对象间的活动并发执行起来。

2.2.1 异步消息传递

在面向对象模型中，对象间通过相互发送消息进行合作，共同完成系统的功能。如果消息发送者在发送消息后不需消息处理结果，或者不立即需要处理结果，消息发送者不必等待消息接受者接收和处理消息即可继续执行下去，只有在消息发送者需要消息的处理结果时才等待。消息接受者对消息进行缓存，在适当的时

候处理之。采用异步消息发送需要解决的两个问题是：

(1)消息处理者如何返回处理结果？
(2)消息发送者如何得到处理结果？

对于问题(1)的一般解决方案是：在消息中带有发送者的地址(标识)，消息处理者在处理完某消息后，将根据消息中的发送者标识给其发回返回值消息。关于问题(2)的一种解决方案是把对象的消息分成两类：一类是普通消息，另一类是回答(返回值)消息。这样往往要把一个完整的消息处理过程分成若干个子过程，其中一个用于处理普通消息，其它的用于处理返回值消息，从而给程序设计带来麻烦。另一种解决问题(2)的途径是：对象发送消息时，消息中带的不是发送者的地址，而是另一个对象(发送者的代理)的地址[19,91,92]，这个代理对象在消息发送时创建，消息处理结果将送给这个对象，当消息发送者需要消息的处理结果时，将向这个对象发送消息以获得消息处理结果，这时采用同步消息发送。

2.2.2 主动对象

在面向对象模型中引进并发的另一种途径是给对象加一个自己的执行线程，这个执行线程常被称为对象的体，有对象体的对象被称为主动对象[10,19,50]。当主动对象创建时，它的体就立即作为单独的线程开始执行。在对象体的执行中，对象可以接收和发送消息。

目前，关于主动对象的研究大多数是为了把进程的概念引进到面向对象模型中来，用主动对象模拟进程；对象间消息传递采用CSP[43]或Ada[13]中的同步消息传递机制。在采用异步消息传递机制的并发面向对象模型中，系统的并发是由异步消息发送产生，而在基于主动对象的并发面向对象模型中，系统的并发则是由创建新对象产生的。两种模型在描述系统的并发程度上是不同，前者属于一种细粒度(fine-grained)控制，并发程度较高，后者是一种粗粒度(coarse-grained)并发控制，系统的并发程度相对较低。当然，在主动对象模型中也可以采用异步消息传递，从而提高系统的并发度，不过对于并发系统而言，并发度是一个要考虑的问题，另一个需要考虑的问题是系统的可靠性，异步消息传递使得系统的行为难以控制，不利于对程序的理解。关于同步和异步消息传递，论文第三章中还要讨论。

2.2.3 对象内部并发

对象间存在并发，对象内部是否也允许并发？在[89]中，根据对象内部是顺序的、半并发的和并发的，对并发面向对象系统中的对象进行了分类：

(1)顺序对象(sequential objects)：对象以FIFO的结构顺序地从消息队列中选择满足条件的消息进行处理，这时，对象中只有一个执行线程。如ABCL/1[92]和POOL-T[10]中的对象以及Ada中的task[13]。

(2)半并发对象(quasi-concurrent objects)：对象中存在多个执行线程，但在某一时刻只有一个是活动的。如管程(monitor)对象，除了一个线程可以执行外，其它线程都因调用条件变量上的wait而被挂起。

(3)并发对象(concurrent objects)：对象中同时存在多个活动线程，一个线程只有在需要等待使用对象的局部变量(由对象的多个线程共享)或其它条件时，才被挂起。

出于表达能力的考虑，应允许对象内部的并发，因为在现实世界中存在这样的对象，其内部事务存在并发行为。另外，从实现的角度说，如果对象间采用同步消息传递，当对象处理消息时又向其它对象发送消息，在等待消息处理结果时，它就不能再接收受其它消息，特别是当对象递归(直接或间接)发送消息时，立即产生死锁。

如果允许对象内部的并发，必须要考虑：

(1)何时创建新线程。一种方案是当对象接收消息时创建新线程，一个线程处理一个消息；另外一种方案是在对象处理一个消息的过程中创建新线程，这时，新创建的线程承担原消息处理的一部分工作。

(2)保证对象状态的完整性。由于各活动线程可能同时使用对象的成员变量，为了保证对象状态的完整，这些线程必须互斥地使用它们。

(3)消息接收的调度问题。由于允许对象内部并发，对象在接收消息时就可能选取后到的满足条件的消息进行处理，这样就会使得先到的消息后处理，甚至永远得不到处理，即出现饿死(starvation)现象。如典型的readers/writer问题，如果调

度策略设计的不好，就会出现多个read消息"合谋饿死"write消息的情形。因此，对象在进行消息接收的调度时，应考虑调度策略的公平性。

2.3 对象的并发控制

在并发面向对象模型中，对象间不存在共享变量但存在共享对象。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，如果对方法的并发调用不进行同步控制，将会造成对象状态的混乱，从而导致整个系统行为的不正确。

如何在对象模型中进行并发控制，一般存在两种途径，即：正交途径和集成途径。在正交途径中，系统的并发执行与对象分开考虑，对象本身不提供并发控制，而是在对象的方法中显式地采用传统的同步机制(如:信号量等)来控制系统的并发与同步，并发控制可以在消息发送方的方法中进行，也可以在消息接受方的方法中完成。正交途径只是在传统并发程序设计基础上加上了对象这个框架，并没有从面向对象模型本身的特点来考虑并发，一个对象的并发控制要依赖于其它对象的配合，从而不符合对象所具有的自治性与局部性特点，并且，在对象的方法中进行并发控制会使得对象类的并发属性难以继承。

集成途径把并发与对象结合起来考虑，对象本身提供并发控制，即，对象对其方法的并发调用进行同步控制。对象在接收消息时，将根据自身的状态决定是否处理该消息。

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，而不应由环境对其行为有过多的假设。在设计对象类时应考虑外部环境可能对其方法的并发调用，这样设计的对象类可适应于各种环境(顺序和并发)。因此，集成途径是从对象模型本身的特点出发考虑对象模型的并发控制，它不仅使得设计的对象(类)便于使用，而且还有利于对象类的重用。

2.3.1 集中控制和分散控制

对象的并发控制从方式上可以分为集中控制和分散控制。集中控制是指把对象的并发控制集中在对象(类)的某一处进行描述。例如，用主动对象的体控制消息的接受就属于一种集中控制。另一种集中控制方式是采用一种较抽象的形式(如：路径表达式)来描述对象的并发行为，当对象接收消息时，根据这个抽象的并发描述决定是否处理所接收到的消息。采用集中控制方式来实现对象的并发控制要求：在设计并发对象类时，必须对对象的并发行为有一个整体的了解并给出完整的描

述。

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，对象无条件地接受消息，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。另一种分散控制方式是在对象类中给每个方法定义一个执行条件，当对象接收到消息时，将根据相应方法的执行条件来决定是否处理该消息，对不满足执行条件的消息让其等待。采用分散控制方式进行对象的并发控制不必要求在设计并发对象类时掌握对象并发行为的全局知识，而是针对每一个消息处理方法来考虑对象的并发控制。

2.3.2 隐式控制和显式控制

对象的并发控制除了可分为集中控制和分散控制外，还可以分为隐式控制和显式控制。隐式控制是指：在并发对象类的定义中把对象的并发限制作为对象的属性进行描述，并不给出具体控制代码，对象的并发控制代码由编译系统根据对象类的并发属性自动产生，对象在接受消息时隐式地调用这段代码。

在隐式控制方式中，对象的并发属性描述和功能(方法实现)描述可以放在同一个类中，也可以用一些抽象的并发模板类来描述对象的各种并发模型，在定义一个并发对象类时，这个类本身只给出消息处理方法的实现，不考虑并发，但在类的定义中必须指定继承哪一个并发模板类，由这并发模板类来规定所定义的并发对象类的并发行为。

显式控制是指：在对象类的定义中明确地给出并发控制代码，这些代码可以放在对象各方法的实现中，也可以用专门的设施(如：对象体)来实现。

隐式控制的好处在于可以把对象的并发控制与对象方法的实现分开考虑，在定义并发对象类的子类时，可以分别继承父类的并发控制和方法的实现。隐式控制的不足之处在于它限制了系统的表达能力，使得对象的并发控制不灵活。

显式控制的优点是使得对象的并发控制描述比较灵活，缺点是：它使得对象的并发控制缺乏系统性，容易出错，另外，在定义并发类的子类时，如果要修改父类的并发控制，就必须连父类方法的实现一起修改，造成不必要的方法重定义。

2.3.3 对象并发状态的表示

对象是数据和操作的封装体，数据存放在对象的局部变量中，对象的状态由对象所有局部变量在某一时刻的取值来表示。在并发环境中，还要考虑对象的并发状态描述问题，因为，对象的并发控制是根据对象的并发状态来进行的。

1. 隐式描述

对象的并发状态往往也隐含在这些成员变量中。例如：对于一个有界缓存对象(bounded buffer)，其状态可由三个成员变量表示：buffer、in、out，存放在有界缓存对象中的元素由buffer表示，buffer中元素的数量由in和out表示。这里，in和out除了表示buffer中元素的个数以外，还表示了有界缓存对象的并发状态。当in=out时，缓存为空，不能接收get消息；当 (in+1) MOD max_len = out时，缓存已满，不能接收put消息。

2. 显式描述

为了更加清晰地表示对象的并发状态，可把对象的并发状态和非并发状态分开进行描述。对象的方法在对象的非并发状态上进行操作，而对象的并发控制在对象的并发状态上进行操作。把对象的并发状态和非并发状态分开可有很多做法：

(1)给对象加上一个路径表达式，用于表示对象的各种并发限制，对象接受消息时将根据该路径表达式来决定是否处理接收到的消息[35]。

(2)用单独的一组成员变量来表示对象的并发状态，这些变量只能由对象的并发控制使用，对象的方法不能对它们进行操作[71]。

(3)把对象的并发状态表示为若干个由对象的方法所组成的集合，这些集合被称为行为抽象[49]，每个集合给出了对象在某一时刻所能接受的消息。例如对于上述的有界缓存对象，其行为抽象可表示为三个集合：{get}、{put}和{get, put}，其中，{get}表示缓存满，{put}表示缓存为空，{get,put}表示缓存不空也不满。

把对象的并发状态描述从对象的非并发状态中分离出来，其好处在于对象的并发行为容易理解，有利于对象的并发控制，并且为解决并发对象类的继承异常(详见2.3.2)提供了一种途径。不足之处在于，它加重了程序设计者的负担，它要求程序设计者必须在二类状态(并发与非并发)上考虑问题，处理不慎易造成二类状态间的不一致。

2.4 并发与继承

继承是面向对象技术的一个重要特性，它是一种信息共享机制，是实现软件重用的一个重要手段。对于并发面向对象模型而言，继承更显得重要，因为并发程序很难设计，如果能利用已有的并发程序代码来设计新的并发系统将会减轻工作量。因此，在面向对象模型中引进并发机制的同时，应该要考虑对继承机制的

支持，采用的并发机制应不损害面向对象模型的继承特性。

然而，很多的研究表明：在面向对象模型中，并发与继承不是两个独立的概念，它们存在相互冲突的性质[59,15,31,62,71,85]。这个问题解决的不好，将会带来继承异常现象[59]，它使得并发对象类难以继承。

由于并发与继承间存在冲突，不少的并发面向对象模型放弃了把继承作为它们的主要语言特性，即使存在支持继承的并发对象模型，但它们对继承的支持程度也是不一样的，这主要取决于它们采用何种并发控制机制。

2.4.1 同步限制、同步代码与同步机制

在并发环境中，处于某状态下的对象不是对发送给它的所有消息都能处理，它必须要根据目前的(并发)状态决定能处理哪些消息，即对象在接收消息处理时必须要满足某种限制，这种限制被称为对象的同步限制(synchronization constraint)。例如对于有界缓存对象，其并发限制是：当缓存为空时只能接收put消息，不能接收get消息；当缓存满时只能接收get消息不能接收put消息。

在对象中为了实现同步限制，必须提供相应的并发控制代码，即同步代码(synchronization code)，这个同步代码必须要与对象的同步限制相一致，否则对象的语义将发生错误。

如何实现对象的同步代码，采用集中控制还是分散控制、隐式控制还是显式控制，实现同步代码的方式被称为对象的同步机制(synchronization scheme)。

2.4.2 继承异常

一个并发对象包括两方面的操作：并发控制和消息处理。在定义并发对象类的子类时，不仅可以继承父类的消息处理方法，而且也应该可以继承父类的并发控制。因为并发控制和消息处理属于对象的两类不同操作，因此在定义子类时应对这两类操作分开进行继承。这样不仅使得并发对象类的描述比较清晰，而且还可以在定义一个并发对象类时从现有的一些非并发对象类中继承消息处理方法，然后再加上一些并发控制，从而可以提高软件重用范围。

然而，如果引进的并发控制机制不恰当，将会损害面向对象模型的继承机制。例如，如果把对象的并发控制放在对象的方法中实现，那么，当定义子类时，由于定义了新的方法而产生新的并发限制，从而需要修改父类的并发控制，这样就要对父类的消息处理方法进行重定义，这种重定义有时是很严重的，往往需要对父类的每个消息处理方法都要进行，这样就彻底违背了继承机制的目的，使得通

过继承进行软件重用的机制大打折扣。上述现象在[59]中被称为继承异常。

关于继承异常的研究普遍认为，造成继承异常的原因主要在于所采用的同步机制，并且不同的同步机制所造成继承异常的程度又各不相同。解决继承异常的关键在于，应该把对象的并发控制代码与对象的功能性代码(对象的方法)分开进行设计。

2.5 对象与进程

在面向对象模型中引进并发，不可避免地要涉及到进程，因为进程是目前进行并发程序设计的主要概念。在一些并发面向对象的研究中，首先考虑的是采用何种面向对象成分描述进程。目前，在面向对象模型中描述进程的典型做法是引进主动对象的概念，用主动对象描述进程，用对象体描述进程的执行。

虽然对象与采用消息传递机制进行通信的进程有很多相似之处，例如：它们都有状态，都能接收消息，但是，进程与对象是有本质区别的。首先，进程是一个实现级的概念，而面向对象软件模型是为了对客观世界中的活动进行自然模拟，其中的对象对应着一些有形或无形的实在对象，如果在其中混入一些实现方面的具体内容，将会影响模型描述问题的清晰度。其次，进程有自己固定的执行次序，即，先做什么，再做什么；何时接收消息，何时发送消息，等等，都已在进程中明确规定了，即使在进程中引进了不确定性成分，如：选择式通信语句及卫士命令[43]等，但其总体执行行为已经确定。而对象的执行行为是由对象的使用者决定。一般来说，对象向外提供很多服务，何时及采用什么次序来使用这些服务是由使用者根据需要决定，而不应该由对象对其使用者的行为有过多的限制，当对象处于不适合于提供某种服务的状态时，可以让使用者等待。如果用主动对象来描述进程，这样就规定了对象的执行行为，同时也限制了使用者使用该对象的方式。最后，在对象体中实现对象的并发控制将会带来继承异常。

2.6 现有系统介绍

目前，关于并发面向对象模型的研究有很多，为了对这些研究有一个整体的了解，以及考察它们对上述问题的解决方案，从中获得启发，本节选择一些典型的并发面向对象系统进行介绍。

2.6.1 Actor 模型

actor模型是一种基于函数式程序设计的并发对象模型，它把对象构造成其行为是消息的函数。actor模型首先是由Hewiit[41]提出的，后来Agha[3]对其进行了进一步的发展。

在actor模型中，对象被称为actor，它是自含的、交互的和独立的软件构件，各actor之间通过异步消息传递进行通信。一个actor由两部分构成：信箱和行为，信箱用于存放接收到的消息；行为用于对消息进行处理。信箱由地址来标识，一旦一个actor创建后，其信箱地址永远不变，但其行为在处理完一个消息后将发生变化，从而产生另一个行为，actor将以这个新的行为处理下一个消息。

构成actor的基本操作有三个：

(1)create: 根据一个行为描述和一组参数创建一个新的actor，其中，参数中可包括现有一些actor。当一个actor处理消息时，它可通过创建一些新的actor来完成原消息处理的一部分工作。这里的create操作对于并发程序设计就相当于lumda抽象对于顺序程序设计，它在并发计算中扩充了动态资源创建能力。

(2)send to: 向另一个actor发送消息，即把消息放入另一个actor的信箱中，它构成了actor模型的基本通信原语，由于它是异步发送，因此actor之间可以并发执行。这里的send to操作相当于异步的函数应用。

(3)become: 用一个新的行为代替当前的行为，它使得actor能够描述历史敏感行为。因为actor是基于无状态的函数概念，为了能描述actor的状态变化，在actor中用行为替换来实现，而不是采用传统的赋值语句。当然，对于一个纯函数式actor来讲，其替换行为与原来的行为是一样的。在actor处理消息的同时，它还计算出用于处理下一个消息的行为，一旦得到了一个新的行为，actor就能处理下一个消息了。因此，在actor模型中，只要become操作不是消息处理的最后一个操作，一个actor的内部就可以存在一种类似流水线(pipeline)的并发，即，下一个消息的处理与上一个消息在执行become操作之后的处理并发进行。

actor模型强调的是所采用的程序结构应能简化对程序的推理(reasoning)，这里的推理不仅仅指程序验证(verification)，更重要的是对软件特性的理解(understanding)，这可通过形式的或非形式的推理来获得。虽然，actor模型在实际使用时还存在很多限制，如：actor的状态表示及对继承机制的支持(actor模型不支持继承)等，但它的一些思想对后来的一些并发面向对象模型产生了很大影响，很多较有影响的并发面向对象模型(如：ABCL/1、ACT++等)都是在actor模型的基础

上进行改进得到的。

2.6.2 ABCL/1

ABCL/1是Akinori Yonezawa等人设计的一个面向对象并发程序设计语言[92]，用于描述他们所提出的一种并发面向对象模型(以下简称为ABCL/1模型)。ABCL/1模型吸收了actor模型的一些思想，但在很多方面又与actor模型不同，例如：ABCL/1兼有作用式(applicative)和命令式(imperative)两种程序设计风格，从而使得传统的程序设计者能较容易地阅读和书写ABCL/1程序。

在ABCL/1模型中，对象被看成是自治的信息处理实体。在一个对象中，不仅可以有表示对象状态的局部变量，而且还给出了对象的行为描述，称为script，它描述了对象接收消息的条件限制和对消息的处理。在某一时刻，对象总处于三种模式之一：休眠(dormant)、活动(active)和等待(waiting)。对象一开始处于休眠模式，当接收到满足其某个并发限制的一条消息后，对象就变成活动模式，开始处理消息。对象处理完一个消息后，如果没有后继消息到达，就又变成休眠模式。当对象在处理一个消息的过程中需要某个条件的出现(往往是等待另一个消息的到达)时，它将进入等待模式，一旦所等待的消息出现后，对象又恢复到活动模式。

在ABCL/1中，对象间采用异步消息传递，消息接受者对消息进行缓存。为了简化程序设计，ABCL/1作了消息传送次序保留假设(Tansmission Ordering Preservation Assumption)，即，从同一个对象发送给另一个对象的两条消息将以与发送次序相同的次序到达消息接受者。如果没有这个假设，消息处理算法将变得十分复杂。

为了描述并发系统中各组成部分之间的各种交互模型，在ABCL/1中，把消息传递分成两种模式：通常模式(Ordinary)和直达模式(Express)，每个对象用两个消息队列分别存放以不同模式发送来的消息。

(1)通常模式：假设一个对象接受到一个以通常模式发送来的消息，并且通常消息队列为空，这时将根据对象所处的模式分三种情况考虑：

• 休眠模式。如果消息可接收(满足对象的并发限制)，则对象进入活动状态，处理该消息；否则，该消息将被废弃。

• 活动模式。消息放入通常消息队列的尾部。

• 等待模式。首先检查该消息是否是所等待的可接收的消息，若是，则对象进入活动模式，处理该消息；否则，该消息放入通常消息队列。

当处理完消息后，如果通常消息队列为空，对象则进入休眠模式；否则，检查队列中第一个消息是否可接收，如果可以接收，对象则继续处于活动状态并处理该消息；否则，该消息将被废弃，这时，将继续检查队列中的第一个消息，一直到队列为空。如果对象在处理某个通常消息时进入等待模式，它将从队列中选取第一个可接收的消息，若存在这样的消息，则对象进入活动状态处理该消息；否则，对象继续处于等待模式。

(2)直达模式：当对象接受到一个以直达模式发送来的消息时，如果对象正在处理一个直达消息，则接受到的消息放入直达消息队列尾部；否则，检查该消息是否可以接收，若可以，则处理该消息，这时，如果对象正在处理一个通常消息，则这个通常消息的处理将被挂起，一直到直达消息处理完。

直达消息传递模式的好处在于：对象可以临时挂起正在处理的(通常)消息，以处理一些紧急事务(直达消息)。

在面向对象系统中，对象间进行交互的唯一方式就是消息传递。为了利用消息传递潜在的同步特性，ABCL/1又把消息传递分成三种类型：过去(past)、现在(now)和将来(future)。

(1)过去消息传递：当一个对象向另一个对象发送一个消息后，前者不必等待后者接收和处理该消息，一旦消息发出后，消息发送者就继续执行其它操作。这种消息传递类型比较适合于消息发送者不需要消息处理结果的情形。

(2)现在消息传递：当一个对象向另一个对象发送一个消息后，前者必须等待后者接收并处理了该消息才能继续执行其它操作。这种消息传递类型类似于函数/过程调用，但又与它们不同，不同之处在于：消息处理者不必等到消息完全处理完才返回，它可以在一旦计算出返回结果后立即返回，然后再继续处理一些善后工作，这样可以减少消息发送者的等待时间。现在消息传递隐含了一种同步，不过，递归的现在消息传递将导致死锁。

(3)将来消息传递：当一个对象向另一个对象发送一个消息后，如果不立即需要消息处理结果，则前者不必等待消息处理，可以继续执行以后的操作，当需要消息处理结果而结果还未产生时才等待。在ABCL/1中，采用future变量机制来获得

将来消息传递的返回结果，即，在消息传递时，指定将消息返回结果传给一个特殊的变量(future变量)，今后需要返回结果时，将从这个变量中获取，如果future变量的值还未产生(对应消息处理还未返回结果)，则等待。

ABCL/1中的对象是顺序对象，对象内部没有并发，在某一时刻，对象只能处理一条消息。另外，在ABCL/1中也没有对并发对象类的继承作专门的考虑。

2.6.3 POOL

POOL是一个系统化的并行面向对象语言[9,10,7]，其设计目标在于：充分利用面向对象模型所具有的描述并行系统的能力，解决在多处理器系体系结构上进行有效的并行程序设计问题。

在POOL中，对象既是程序的基本保护单位，同时它又构成了系统的基本并发单位。系统的并发执行是通过给对象加一个体(body)来实现的，对象体的执行不需要由消息来激活，当一个对象创建时，其体就自动开始执行。对象间采用同步消息传递方式进行通信，对象在体中用answer语句显式地控制消息的接收，这与Ada中task的消息控制类似。对象在体中接收了一条消息后，它将调用其某个方法来处理之，这时，对象体的执行被挂起，当消息处理方法执行结束后，对象体重新得到执行，因此，POOL中的对象也是一种顺序对象，对象内部没有并发。另外，如果在设计一个对象类时没有给出对象体，那么系统将自动给其加上一个缺省体，该对象体只是不断地顺序接收消息，然后调用相应方法进行消息处理。在POOL中，把显式地给出了对象体的对象称为主动对象，没有给出对象体的对象称为被动对象。

由于采用同步消息传递，为了减少消息发送者的等待时间，在POOL中，对象不必在消息处理结束时才返回消息发送者，它可以在消息处理方法中任意的地方返回(只要消息结果已产生)，然后继续进行一些后处理。另外，为了解决递归的同步消息传递带来的死锁问题，POOL允许在消息处理方法中直接调用对象的其它方法，而不需要经过对象体的消息接收处理。

与ABCL/1不同，为了保证公平性(fairness)，POOL中的对象只有一个消息队列，它保证了只要某消息出现在队列中，当在对象体中执行的answer语句时，如果其中包含该消息，那么这个消息一定会得到处理。

POOL的设计者Pierre America对继承持下述观点：子类应是父类行为上的特化(行为继承)，它们应具有子类型关系，这样不仅便于静态类型检查，而且也使得程

序容易理解。但是，在POOL中，由于采用对象体来描述对象的并发行为，在定义并发类的子类时往往要修改(重定义)父类的体，以把子类中新定义的消息处理的并发限制考虑进来，这样，就有可能使得子类与父类有不同的(并发)行为。因此，POOL的早期版本POOL-T[10]放弃了把继承作为其主要语言特色，但在其较新版本POOL-I[7]中，继承以一种清晰和安全的方式被引进，其做法是把继承机制与子类型设施分开进行考虑，继承仅仅作为一种代码复用机制，类与子类之间不需要有子类型关系。另外，在POOL-I中，继承只用于对象类的方法，在定义子类时，对象体不继承，因为父类与子类的对象体一般是不同的。

2.6.4 Hybrid

Hybrid[72]是一个并发面向对象语言及其运行系统，其设计目标在于把基本的面向对象概念与并发计算进行集成。Hybrid是一个强类型语言，其中的类型等价定义为结构等价，它不要求等价的类型之间存在继承关系(在Hybrid中类与类型合为一个概念)。

Hybrid中的基本的并发单位是域(domain)，一个域由一个顶层对象("top-level" object)和若干个子对象(subobjects)构成。当创建一个对象时，创建者可以指定该对象属于创建者所在的域或是一个新域中顶层对象。一个域可以处于三种状态之一：活动、阻塞和空闲，当域中的某对象正在执行一个消息处理时，域就是活动的；当域中的某对象调用了其它对象的方法而等待处理结果时，域就变成阻塞状态；如果域既不处于活动又不处于阻塞状态，则它为空闲状态。

在Hybrid中，并发执行是通过向一个对象发送start消息产生的，这时发送者与接收者并发执行。对象间采用远程过程调用(RPC)进行消息传递，一系列嵌套的RPC称为一个活动(activity)。处于空闲状态域中的一个对象可以接受来自任何活动的任何消息；处于阻塞状态域中的对象只能接受返回值或造成其阻塞的活动发送来的消息，这样可以避免递归等待(死锁)；一个域中只能有一个活动的消息处理过程。

对象除了具有其所在域提供的并发控制以外，它还可以有自己的并发控制，这可通过采用延迟队列(delay queue)机制在其类定义中进行设计。延迟队列是对象的实例变量，它们可以被打开和关闭。对象的每个方法都可以与一个延迟队列相关联，当对象接受消息时，若相应消息处理方法的延迟队列处于关闭状态，则消息接收被延迟，直到其延迟队列被其它消息处理打开。延迟队列可实现消息处理的局部延迟(local delay)，即消息响应被延迟。

由于延迟队列机制无法实现依赖于消息参数的局部延迟，因此，Hybrid提供了

一种代理调用机制(delegated call)。当采用代理调用机制进行消息传递时，调用者线程被挂起，调用者所属的域变成空闲状态，这样，调用者及其所在域中的其它对象就可以响应其它消息。当代理调用返回后，如果调用者所属域中没有活动的消息处理，则原来被挂起的线程恢复执行。代理调用机制可以实现消息处理的远程延迟(remote delay)，即在消息处理过程中需要等待某条件的产生，这时等待发生在其它对象那里，消息处理者仍然可以响应其它消息。

2.6.5. DRAGOON

DRAGOON[11]是一个基于Ada的面向对象语言，主要用于描述嵌入式应用。DRAGOON中除了保留了Ada的很多特色和风格以外，还引进了面向对象的一些成分。与Ada一样，DRAGOON既可以作为一个编程语言，又可以用于描述软件设计。

在DRAGOON中，并发描述是通过多继承机制来实现的。在定义一个并发对象类时，必须指出它从两个类继承而来：一个是具体的顺序对象类，另一个是抽象的并发描述类。其中，顺序对象类给出了并发对象类的功能描述，并发描述类给出了并发对象类的并发行为描述。在并发描述类中，用带有历史函数的逻辑表达式来描述若干抽象操作间的并发限制，在继承它的并发对象类中，必须给出这些抽象操作与类中的具体操作之间的对应关系。并发描述类可以用于多个具有相同并发行为的并发对象类，但它本身不能直接用于创建对象。

在DRAGOON中，消息传递采用远程过程调用方式，对象内部允许并发。另外，对象可以有一个体(在DRAGOON中被称为thread)，但与其它基于主动对象的系统不同的是：体的执行不是隐式(自动)的，它必须由对象的使用者显式地调用对象的一个方法start来启动它。由于对象的thread不受并发描述类中所给出的并发限制的约束，它可以不加限制地使用对象的成员变量，这样就有可能与对象方法的执行发生冲突，DRAGOON对这个问题的解决办法是：用内部(局部)对象来实现对象的thread与方法所共享的成员变量，因为，在DRAGOON中，对象的局部对象也可以是并发的(这与其它的一些系统不同)，因此，可以用并发的局部对象来协调对象的thread与方法执行的冲突。

2.6.6 ACT++

ACT++[49]是C++[82]的一种并发扩充，它采用了actor模型[3]的一些并发特色。在ACT++中，对象分成两类：actor和被动对象。actor是一种主动对象，具有并发控制能力，它顺序地接收消息处理；被动对象就是通常的C++对象，它只能作为actor

的局部对象，因此不能对其并发操作。actor对象由一个预定义的actor类的子类所创建。

ACT++的主要通信机制是远程过程调用，系统的并发执行是由reply和become操作实现的。在一个方法的执行中，如果执行了reply操作，则方法的调用者和reply之后的操作并发执行。当方法中执行become操作时，其语义同actor模型，即，对象将以一种新的行为准备接收下一条消息，原先的消息处理仍可继续执行。

在ACT++中，对象根据当前的行为描述来选择消息，这里的行为描述是用对象某一时刻所能接收的方法集(称为行为抽象)来表示的，become操作给出了对象下一个状态所能接收的方法集。这种行为抽象思想可以看作是对顺序面向对象语言的数据抽象机制在时序方面的扩充。另外，在定义子类时，可以继承和重定义父类的行为抽象，采用行为抽象来描述对象的并发控制为解决继承异常现象提供了一种途径，但不是彻底解决[59]，因为下一个并发状态还是要在方法实现中用become操作给出。

ACT++中的行为抽象是不能对其操作的，在另一个类似的系统Rosette[85]中，用Enabled-set表示行为抽象，与ACT++不同的是，Enables-set可以进行操作，这样就进一步加强了行为抽象的表达能力。

2.6.7 Smalltalk 和 ConcurrentSmalltalk

1. Smalltalk

Smalltalk[36]设计主旨不是针对并发计算的，其目的是充分展示面向对象程序设计思想，它对后来的面向对象技术的研究产生了巨大影响。

从整体上讲，Smalltalk描述的是顺序面向对象模型，其中的消息传递就是(远程)过程调用，对象本身不考虑并发控制。虽然Smalltalk提供了并发描述机制，但它是采用对象以外的概念来实现的。

在Smalltalk中，并发行为的产生是通过在一个消息处理中向一个嵌入在消息处理代码中的分程序(block context)发送fork消息，从而使得这个分程序作为一个进程与原来的消息处理过程并发执行。例如对于下面的代码片断：

```
.....
[...(I)...] fork.
...(II)...
```

.....

(I)与(II)并发执行。

由于在由fork产生的分程序进程中，可以使用原消息处理方法的局部变量、对象的实例变量、类变量和全局变量等，因此，这些变量就构成了进程间的共享资源，在Smalltalk中，进程间采用共享变量进行通信，用信号量实现进程间的同步。另外，在分程序进程中也可以向已有的对象发送消息，这样，就存在共享对象的情形，因此，一个对象可能面临环境对其方法的并发调用，而Smalltalk中的对象是没有自动的自我保护能力的，必须由消息发送者或消息接受者给出显式的同步控制，在Smalltalk中，这往往也是通过信号量来实现的。

2. ConcurrentSmalltalk

在Smalltalk中进行并发程序设计，设计者必须在两个不同的概念(对象和进程)上考虑问题，这不仅加重了设计者的负担，而且也损害了对系统的描述能力和可理解性。

ConcurrentSmalltalk[91]是一个基于Smalltalk并与之保持向上兼容的面向对象并发程序设计语言，该语言的设计宗旨是在一个概念层次上描述并发问题，它把Smalltalk中的对象和进程合成一个概念：并发对象。

在ConcurrentSmalltalk中除了保留了Smalltalk原有的成分以外，还引进三个新的成分：

(1)异步方法调用。作为一种通信方式，它具有下述特性：

- 单向通信。消息发出后，消息发送者不需等待回答消息即可继续执行下去。
- 无控制转移。消息发送者与消息接受者并发执行。
- 消息接受者有权选择消息。消息接受者进行并发控制。
- 消息发送者与接受者间存在主从关系。消息处理者处理消息后必须给出回答。
- 消息接受者在给出回答消息后仍可继续执行。

(2)同步机制。

为了在进行异步方法调用时能获得返回结果，ConcurrentSmalltalk引进了一种

CBox类型的对象，它类似于ABCL/1中的future变量。作为异步方法调用的一种同步设施，CBox对象在进行异步方法调用时创建，用于处理返回结果。当消息发送者需要返回结果时，可通过向相应的CBox对象发送receive消息，如果返回结果已产生，则得到结果，否则将等待。

(3)原子对象。

Smalltalk中的对象是没有并发控制能力的，当向共享的对象发送消息时，为了保证互斥，消息发送者必须提供同步控制，在Smalltalk中，这是通过信号量来实现的。用信号量这种低级同步机制进行并发控制不仅容易出错，而且也会使得程序难以理解(必须对出现在许多对象中的所有P和V操作进行分析)。

在ConcurrentSmalltalk中，为了更好地进行并发控制，引进了另一种对象：原子对象(atomic object)。原子对象具有并发控制能力，它以FIFO的次序顺序地接收和处理消息。为了解决在消息处理中向自己发送消息(通过伪变量self或super)造成死锁问题，ConcurrentSmalltalk中用局部过程调用来实现向self或super发送消息。另外，为避免一个原子对象向另一个对象发送分程序上下文(block context)所造成的死锁问题，在ConcurrentSmalltalk中把分程序上下文的计算作为其消息处理方法的一部分来实现。

原子对象保证了消息接收和处理的互斥性，但消息间的条件同步控制往往要放在各消息处理方法中进行，并且还需要其它对象的配合。例如，当一个对象在处理一个消息时发现：目前状态下它不能处理该消息，必须等它处理了另一个消息后才能处理之。在ConcurrentSmalltalk中可以有两种方式解决这个同步问题：

(1)消息接收者终止该消息的处理，并返回出错信息给消息发送者；消息发送者在适当的时候重新向其发送该消息，这种重新发送可能要重复多次，一直到消息接受者正常处理了该消息。

(2)在方式(1)中，为了获得一条消息的正常处理结果，消息发送者往往处于一个忙式等待(busy waiting)的循环中。为了避免这种忙式等待，也可以采用另一种方式来解决上述的同步问题，即，消息接收者在终止消息处理前，先记下该消息的发送者和所等待的条件，然后返回出错信息给消息发送者；当消息接受者处理了其它一些消息使得先前被终止的消息所需要的条件得到了满足后，再向该消息的发送者发送一个消息让其重新发送原来的消息。当然，在这种方式中，必须要求消息发送者提供一个重新发送的消息处理。

从上述描述可以看出：不管采用方式(1)还是方式(2)，在ConcurrentSmalltalk中，一个对象的并发控制必须借助于其它对象的配合，并且，在方式(2)中，当一个对象所能处理的消息种类很多时，条件同步控制将变得十分复杂。另外，继承异常问题在Smalltalk和ConcurrentSmalltalk是很严重的。

2.6.8 Eiffel、Eiffel//和 CEiffel

Eiffel[63]是由ISI的Bertrand Meyer设计的一种面向对象程序设计语言，断言机制是其一大语言特色，它体现了设计者所倡导的一种软件构造思想：Design by Contract(根据契约进行设计)，这个思想强调的是：义务(obligations)和受益(benefits)，即，当一个对象(客户)调用另一个对象(服务者)的某方法(在Eiffel中称为Routine)时，客户对象的义务是应保证服务者对象相应方法的前置条件成立，服务者对象的义务是在执行完方法后应确保该方法的后置条件成立；在双方尽了各自的义务后，客户对象的受益是得到一个所需要的结果，服务者对象以得到一个不变式作为其受益。

Eiffel的初始设计不是针对并发的，后来各种并发扩充方案相继出现[64,19,56,50,38,45]，包括设计者本人也对Eiffel进行了并发扩充。下面将对Eiffel的几种典型的并发扩充进行介绍。

1、SCOOP

Bertrand Meyer在其<Object-Orient Software Construction>一书的第二版[65]增加了一章，专门介绍了对Eiffel的并发扩充，其中的一些思想发表于设计者早先的一些文献中[64]。

在Meyer对Eiffel的并发扩充方案中，把其所引进的并发机制称作为SCOOP，其设计标准主要有：

- 机制的简洁性(SCOOP一词中的S就表示了这一个标准：Simplicity)。
- 充分支持继承等面向对象技术。
- 与Design by Contract设计思想保持一致。
- 程序的可证明性。
- 支持多种并发计算模式。
- 支持对非并发软件的重用，等等。

在SCOOP中，并发机制的简洁性构成了其主导设计思想。如果并发机制设计的很复杂，不仅会使得其中的一些成分与面向对象模型中的已有成分相重复，而且也会造成冲突。SCOOP的简洁性体现在：它只引进了一个新的关键词**separate**，当一个对象x被定义成：

x: separate SOME_TYPE

或

x: SOME_SEPARATE_TYPE

则表示x将在不同的逻辑处理器(processor)上执行，这里，设计者用processor一词来区别于物理上的CPU。processor是一个抽象的概念，它可以是网络上的一台计算机，也可以是操作系统中的一个任务(task)或任务中的一个线程(thread)。当调用具有separate属性的对象的方法时，调用者与被调用者并发执行。

在SCOOP中，存在以下几种同步控制：

(1)按需等待(wait by necessity). 当调用具有separate属性的对象的方法时，调用者与被调用者并发执行，调用者只有在需要结果时才等待，其实现类似于ABCL/1中future变量。

(2)为了实现对具有separate属性对象的互斥使用，可以把separate对象作为实参传给某个具有separate属性形参的方法，当调用这个方法时，如果相应实参所代表的separate对象已被他人占用，则调用者等待，否则，调用者占有该对象，被调方法继续执行。如果一个方法有多个separate属性的形参，只有当相应的每个实参所代表的separate对象均未被占用时，该方法才能得以执行。

(3)当某方法的前置条件中包含有方法的separate属性形参时，则调用者等待该条件满足。

(4)对象中只允许有一个方法执行(顺序对象)。

(5)允许高优先级的客户对象中断(interrupt)一个对象中正在执行的方法，同时在被中断的客户对象中产生一个异常，由该客户对象进行纠正处理，比如，适时地重试。

2、Eiffel//

Eiffel//[19]是Denis Caromel设计的一种并发面向对象模型，它是Eiffel的一种并发扩充。在Eiffel//中首先考虑的是如何描述进程。为了把进程概念引进到面向对象模型中来，Eiffel//采用主动对象来描述进程。在Eiffel中提供了一个类：PROCESS，用该类或其子类创建的对象就是进程，从而把进程概念结合到对象概念中。

在PROCESS类中提供了一个方法：Live，它相当于POOL中的body，子类中可以对它进行重定义。一旦一个进程对象被创建，其Live方法就开始执行，当Live执行结束后，进程对象就消亡了。Live的主要作用是控制消息接收，在PROCESS类中给出了Live的缺省实现：FIFO。

在Eiffel//中并不是每一个对象都是进程，只有进程对象才具有并发控制，因此，为了解决多个进程对象共享同一个对象所带来的同步问题，Eiffel//规定：非进程对象不能被共享，它们只能作为进程对象的私有(局部)对象，供一个对象使用。另外，当一个非进程对象作为方法的参数出现时，传递的不是该对象的引用(reference)，而是它的一个副本(copy)。进程对象作为参数传递时，传递的是它的引用。

当一个对象向另一个发送消息时，如果消息接受对象是非进程对象，则消息传递具有通常的过程调用语义(同步消息传递)，否则，将在消息接受对象中产生一个异常，消息接受对象目前的消息处理被中断，这时，消息发送者与接受者进行信息交换(handshake)，其中，一个类型为REQUEST的对象将从消息发送者传给接受者，该对象记录了所传递的消息和相应的参数，它被放入消息接受者的消息队列(异步消息传递)，然后，消息发送者与接受者继续执行各自的操作。消息接受者今后将在其Live过程中从消息队列获取消息(REQUEST类型的对象)进行处理，消息处理结果将以与上述相同的方式发送该消息的发送者。因为消息队列属于对象接受消息和选取消息这两个操作的共享资源，为了保证互斥，当对象从消息队列中选取消息时，它临时把自在变成忽略(ignore)模式，不再接受消息(这也是通过异常机制来实现的)；选取消息后，对象再变成继续模式(continue)。

在Eiffel//中为了解决异步消息传递的同步问题，也采用了与ABCL/1中的future变量相类似的方式得到消息处理结果。

3、CEiffel

CEiffel[56]是Eiffel的另一种并发扩充 方案，这个方案着重强调了并发与继承的集成，即，所引进的并发描述应有利于继承。在Eiffel的诸多并发扩充方案中，CEiffel的做法与众不同，它把扩充的并发描述成分以一种特殊的注释形式(称为并发注释)加入到Eiffel程序中，这样设计的一个程序如果用Eiffel编译器来编译，得到的是一个顺序的可执行程序，但如果用CEiffel编译器对它进行编译，得到的将是一个并发的可执行程序。

在CEiffel中，不同的并发控制成分用不同的并发注释来表示：

• 相容注释(compatibility annotation)。用 --||-- 或 --|| 方法名表 -- 来表示。

相容注释可以出现在类定义的首部，也可出现在方法定义的首部。当相容注释出现在类定义的首部时，如：class A --||-- ，表示该类中的各方法间可以不受限制地并发执行；当相容注释出现在方法首部时，如：m0(...): T is --|| m1,m2,...-- 和 m0(): T is --||-- ，前者表示m0可以与m1、m2、...并发执行，后者表示可以有多个并发执行的m0，并且m0可以与其它具有这样注释的方法并发执行。相容注释是对称的(symmetric)，但不一定是传递(transitive)或自反(reflexive)的。当一个消息到达时，如果其相应处理方法与正在执行的某个方法不相容，则该消息处理必须等待，一直到所有与之不相容的方法执行完毕。当一个方法执行结束时，若有几个等待的消息处理可以执行，则根据它们到达的次序来决定选择次序。对局部方法调用不进行相容性检查。

• 延迟注释(delay annotation)。用 --@-- 表示。

延迟注释可以出现在方法的前置条件中，它往往把前置条件中的断言从句分成两部分，前一部分称为checker，后一部分称为guard，如：put(item: T) is require item /= Void; --@-- length < size do ... end; 其中，item /= Void为checker，length < size为guard。当对象接收了一个满足相容性的消息处理后，如果checker不成立，则产生一个异常，消息被拒绝；如果guard不成立，则消息处理必须等待(被延迟)，否则，执行消息处理。另外，延迟注释也可以出现在后置条件中，用于实现诸如调度(scheduler)之类的功能。

• 自治注释(autonomy annotation)。用 -->-- 表示。

自治注释可以出现在方法首部，这时，相应方法不应有参数和checker，它表示一旦对象被创建，该方法立即就隐式地被调用，如果其guard成立，则方法自动开始执行，每当带有自治注释的方法执行结束时，它将再次被隐式地调用。带有自治注释的方法一般作为私有过程，不向其它对象开放。自治注释是CEiffel中产生并发执行的机制之一。

• 异步注释(asynchrony annotation)。用 --v-- 表示。

当异步注释出现在某方法定义首部时，它表示该方法与调用者并发执行(异步消息传递)。在调用一个异步方法时，只要其checker成立，调用者就可继续执行。

异步注释不仅仅用于过程，它也可以用于函数，对于异步函数，CEiffel中也采用了类似于future变量的机制来获得结果。异步注释是CEiffel中产生并发执行的另一种机制。

• 控制注释(control annotation)。用 --!-- 表示。

当用一个类来给出对象的类型(在Eiffel中类也是类型)时，如果在类名的后面加上了控制注释，如：o: A --!--；则表示所定义的对象o是一个受控对象(controlled object)，它具有A中所有并发注释所给出的并发控制；否则，o是一个顺序对象()，A中所有并发注释所给出的并发控制对它不起作用。如果A中没有并发注释，则对于受控的A类对象o，CEiffel编译器自动把它实现成原子对象(顺序处理消息)。

CEiffel中的继承机制定义如下：

• 相容注释。子类方法的相容注释中可以出现祖先类中的方法名；如在子类中重新定义了祖先类中的某方法，则必须在子类中显式地给出该方法的相容注释，即子类中不继承祖先类中相容注释的对称性；在多继承情况下，当一个子类继承了两个祖先类中的方法时，如果这两个祖先没有公共的祖先，则继承的方法是相容的，否则，是不相容的。如果子类从同一个祖先继承了两个方法，则这两个方法保持原来的相容性。

• 延迟注释。子类方法继承祖先类方法的延迟注释。当在子类方法中给出了新的延迟注释时，如果是在前置条件中，则条件减弱(与祖先类中相应方法的延迟注释是相'或')；如果是在后置条件中，则条件加强(与祖先类中相应方法的延迟注释是相'与')，这是Eiffel的继承原则。

• 自治注释。子类方法继承祖先类方法的自治注释，但在子类中可以重新定义方法的自治注释。

• 异步注释。继承机制与自治注释相同。

CEiffel所采取的并发扩充途径不仅仅针对Eiffel，它同样也适合于其它一些面向对象语言。

2.7 小结

本章从并发系统的一般模型出发，给出了在面向对象模型中引进并发所面临的各种问题，其中包括面向对象模型的并发执行、对象的并发控制以及对继承机制的支持。在提出这些问题的同时，本章还给出了一些可能的解决方案，这些方案各有长处和不足，尤其是在与面向对象模型相容和对继承机制的支持方面。本章还对对象与进程的关系进行了讨论，结论是：进程是一个实现级的概念，对象模型不必对进程进行模拟。为了对目前在并发面向对象模型方面的研究有一个整体的了解，本章还介绍了一些典型的并发面向对象模型，作为论文研究的参照。

第三章 基于并发对象的并发面向对象模型

在第二章中，论文给出了在面向对象模型中引进并发描述应解决的各种问题，并对目前的一些并发面向对象模型进行了介绍，从这些系统所采取的方案可以看出，它们各有侧重点，有的强调表达能力和系统的并发度，而另一些则着重考虑系统性和可靠性。但在并发与面向对象模型的集成方面，大多数系统没有给出很好的解决方案，所采取的并发机制往往与面向对象模型的一些特点不一致，从而损害了面向对象模型为软件开发所带来的好处。

本章在分析和研究了现有系统的基础上，从面向对象模型本身的特点出发，提出了一种基于并发对象的并发面向对象模型。本章首先从对象系统的并发执行、对象间的通信、对象的并发控制以及对继承的支持等四个方面来介绍论文所提出的并发面向对象模型。然后，为了说明所提模型的具体应用，给出了采用该模型所描述的一些典型的并发实例。最后，论文把本章所提模型与其它的一些并发对象模型进行了比较。

3.1 对象系统的并发执行

对象是面向对象系统的基本组成单位，在并发环境中，对象又构成了自然的并发单位，并发面向对象系统的并发行为是由于对象之间存在并发执行的操作而产生的。对象之间并发操作的产生可以是由异步消息传递所引起，也可以是由对象自己的私有执行线程所产生。作者认为，后者更能反映客观世界中的并发活动，因为，对象不都是被动地提供服务，它们中的一些除了能向外提供服务外，还应该有自己的事务处理，否则，服务请求(消息)从何而来？因此，在本论文所提出的并发面向对象模型中，系统的并发执行是由对象的私有执行线程所产生，在对象私有线程的执行中，可以向其它对象发送消息，请求服务。

3.1.1 对象的私有执行线程

为了描述面向对象系统的并发执行，论文在对象内部引进了私有执行线程，不同对象的私有线程之间并发执行。对象的私有线程由对象自己根据需要进行创建，它们用于对象内部的事务处理。

在通常的基于主动对象的系统中，用对象体(body)来实现对象的私有执行线程，每当有对象体的对象创建时，其对象体就作为一个独立的线程开始执行，对

象体执行结束，对象就消亡了。

本论文所采取的做法与基于主动对象的系统有很大的不同，这主要体现在下面两个方面：

(1)私有线程何时产生。

在本论文的并发对象模型中，对象的私有执行线程可以不止一个，其中可以有一个类似于主动对象的对象体，称为初始线程。一旦对象被创建，其初始线程(如果存在)就自动开始执行，但它与主动对象的对象体不同，初始线程执行结束时，对象并不消亡，只有当对象的创建者撤消(显式或隐式)对象时，对象才消亡。对象其它私有线程是在对象的消息处理方法执行线程中创建的，创建操作可以出现在方法实现中的任何地方。一旦创建了一个私有线程，该私有线程就与对象方法所在线程异步(并发)执行。

私有线程形式上相当于对象的一个方法，但又与其它方法有所不同，首先，私有线程不能被其它对象直接调用，只能由对象的方法或对象自己来创建；其次，私有线程没有参数和返回值，它们在对象的局部数据上进行操作；再有，方法执行所在的线程是由其它对象创建的，它们属于其它对象(因为，本论文中的并发对象模型采用远程过程调用，详见3.2)，而私有线程由本对象所创建，它们属于本对象。除此之外，私有线程在对象中的地位与对象方法所在线程完全相同，它们的执行也要受到对象并发控制的限制(详见3.3)，在私有线程的执行中也可以向其它对象发送消息。

(2)私有线程的作用。

在基于主动对象的系统中，对象体的作用是实现对象的并发控制，即控制消息的接收。在本论文所提出的模型中，私有线程不是用于对象的并发控制，而是用作对象内部的事务处理，是实现系统并发执行的一种机制，对象的并发控制以其它方式来实现(详见3.3)。这样，即使一个对象没有私有线程，它仍然具有并发控制能力。而在基于主动对象的并发模型中，一个对象如果没有对象体，它就成了被动对象，其并发控制要由其它对象来完成。

对象私有线程的功能可看作是消息处理方法功能的延伸，为了减少消息发送者的等待时间，在消息处理方法中，执行了一些必要的操作后，就可把控制返回给消息发送者，消息处理的后续工作可以通过启动对象的私有线程来完成。

3.1.2 对象内部的并发

在论文所提出的并发模型中，不仅对象间的活动可以并发执行，而且还允许对象内部存在并发，即，对象内部各线程可以并发执行，这样，除了可以提高系统的并发度以外，还可以避免由于同步消息传递所带来的一些问题，如：递归的同步消息传递将导致死锁。当然，允许对象内部的并发将会给对象的并发控制带来麻烦，因为，对象内部各并发执行线程之间存在共享变量(对象的局部变量)，为了保证对象状态的完整性，这些并发执行的线程应互斥地使用它们。

一个对象的内部是否存在并发对于该对象的使用者来说是透明的，它属于对象的具体实现。一个对象采用何种实现：顺序或并发，应不影响对象的功能(虽然性能会受到影响)，这样，对于使用者来说不会感到有太大的差别，最多是等待时间长短问题。当然，使用者在使用一个对象时，不应该对该对象的实现有任何假设，否则，对象实现的改变将会影响到其使用者。对象内部实现的用户透明性体现了对象的自治性(详见3.1.3)。

3.1.3 对象的自治性

在有些并发对象模型中[19,50,10]，为了把进程结合到对象模型中来，往往把对象分为两类，一类称为主动对象，另一类称为被动对象。主动对象用于实现进程，它们具有并发控制能力，可参与系统的并发操作；被动对象没有并发控制能力，它们只作为其它对象的成员，局部于这些对象，不直接参与系统的并发操作，其并发控制必须由使用者来实现，即，使用者必须保证不对被动对象进行并发操作。

论文认为：上述做法易造成使用上的一些不便。首先，它要求设计者在两个概念中考虑问题，对象的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用了，将使得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又未考虑并发，这样，就可能产生混乱。其次，如果一个(被动)对象的并发控制需要由其它对象来实现，而使用者在使用它时，由于种种原因没有进行并发控制，则将造成灾难性后果。再有，进程属于一种实现级的概念，是对客观并发活动的间接模拟，而对象模型的特点是以一种更加直接的方式来描述客观世界中的活动，如果其中参杂有实现级概念，则会影响到面向对象模型描述问题的清晰性，因此，在对象模型中没有必要对进程进行模拟。

在本论文所提出的并发面向对象模型中，不把对象区分成主动和被动，而是用并发对象来称呼并发环境中的对象，在设计这些对象(类)时，都应考虑并发操作，这体现了对象的自治性，即，对象的并发行为由对象自己控制，而不应由对象的

使用者来对其是主动对象还是被动对象作假设。即使把并发对象用在顺序环境中，也没有害处，关键是，这样设计的对象(类)可适应于各种环境(顺序和并发)，从而有利于软件对象的复用。

3.2 对象间的通信

从本质上而言，面向对象模型属于一种client/server计算模式，一个对象提供一些服务，对象的使用者通过向它发送消息，引起其某方法的执行，该方法执行结束将返回一个处理结果，对象的使用者获得结果后继续执行其它操作。这里的消息传递是一种双向通信。

3.2.1 远程过程调用

在本论文的并发面向对象模型中，消息传递采用了远程过程调用机制，这种通信方式比较符合面向对象模型的特点，并且它使得消息传递实现了对象活动间的一种自然同步，对理解程序的行为有一定的帮助。

在一些并发面向对象模型中，如：actor模型，消息传递采用异步方式。虽然异步消息传递可以提高系统的并发度，但同时也会带来一些问题。首先，它使得程序的行为难于把握，并且，还需要实现同步，即，消息发送者获得回答消息(消息处理结果)，这往往需要对象之间的配合(如通信双方约定采用future变量[92]或CBox对象[91]来实现同步)，从而不符合对象的自治性原则，同时也不利于软件的复用。其次，在异步消息传递系统中还要考虑消息的发送次序和消息的接收次序是否一致的问题。另外，从实际软件开发的角度来看，消息发送对象在发送消息后，一般立即需要返回结果，特别对于存在于表达式中的方法调用尤其如此。即使对于没有返回值的消息传递，消息发送者在发送消息后，今后的操作往往会基于消息已被处理这样的前提，如果消息还未处理(因为是异步消息传递)，则有可能造成消息发送者在消息发送后的操作中的语义发生错误。

虽然，同步消息传递(远程过程调用可由一个发送和一个接收来实现)存在一些问题，如：系统的并发度受到影响，特别是，递归的同步消息传递会造成死锁，但是，如果处理不当，递归的异步消息传递也会产生死锁，如：对象A向对象B异步发送一个消息，对象B在处理该消息时，又向对象A异步发送另一个消息，如果对象A正在等待对象B的处理结果，而对象B在执行到某一步时又等待对象A的处理结果，这样就造成了死锁。

因此，这里的问题不在于同步或异步消息传递，而在于所采用的同步机制以及是否允许对象内部的并发，如果对象在等待时还可以接收消息处理，则可以避免一些死锁问题。关于并发度的问题，如果采用同步消息传递，则可通过其它一些方式来弥补并发度不高的问题，如：允许对象有私有执行线程，当对象处理一个没有返回值的消息时，可以把一部分工作交给对象的私有线程去做，而消息处理可实现成仅仅提交一个请求。

另外，同步消息传递和异步消息传递是可以互相实现的[10]。若采用同步消息传递作为基本通信方式，则可在消息传递时，利用一些临时对象对消息进行中继，从而实现异步消息传递；若基本通信采用异步方式，则可在消息发送后立即等待消息处理结果，实现同步消息传递。不过，异步消息传递在一些情况下会 显得更加自然和灵活一些，特别是对于无返回值的消息传递。在本论文的并发面向对象模型中之所以采用远程过程调用作为对象间的基本通信方式，主要考虑的是它使得程序设计者能够对程序的运行行为有更多的控制，并且与现有的顺序面向对象模型中消息传递机制保持一致。

3.2.2 对象的通信对象

在采用消息传递机制进行通信的并发系统中还存在一个通信信道(communication channel)的指定问题[6]。在本论文的并发面向对象模型中，采用了顺序面向对象模型中的直接命名方式来指定通信对象，即：<对象名>.<方法名>(<参数>)。

对象可以与之通信的对象包括：全局对象、局部对象和方法的参数中所指定的对象，其中，全局对象包括本应用中说明的全局对象和其它应用中说明的全局对象(外部对象)，外部对象的定联是动态的，它可以通过给每个应用加上一个并发配置文件来实现，该配置文件中每个条目具有下述形式：

<外部对象名>：[<计算机名>.]<应用名>.<对象名>

如果<计算机名>省略，则表示外部对象在本机的一个应用中。

在论文中的并发面向对象模型中，对象都应具有并发控制能力，这不仅仅是指全局对象，对于局部对象和方法参数中的对象也应该具有并发控制功能。局部对象的并发性可以用于实现对象内部各并发执行线程间的互斥控制；方法参数可

以是对象的引用，所引用的对象是共享的，可以实现对象间的间接数据交换。

3.3 对象的并发控制

在一个并发系统中，由于竞争共享资源或交换数据(通信)，并发的活动间往往需要进行同步。在并发面向对象系统中，共享资源是作为一些对象的局部数据，由这些对象来对它们进行管理的，对共享资源的使用是通过向管理它们的对象发送消息来实现的，这里，资源管理对象是共享的。非资源管理对象之间的数据交换往往也是通过共享对象来实现的，即使不通过共享对象，它们之间也是通过消息传递来进行通信的。

在顺序面向对象系统中，某一时刻，一个对象只有一个使用者，该对象的行为完全在其使用者的控制之下，即，使用者在操作一个对象(调用其方法)时，该对象的状态完全在使用者的掌握之中。在并发环境中，由于对象可以被共享，这样就存在多个对象对同一个对象的并发操作，如果在设计对象(类)时没有考虑可能的并发，就会造成对象状态的混乱。因此，在设计并发面向对象系统中的对象时，一定要考虑并发操作间的同步控制。

虽然，消息传递隐含着一种同步，但这只是实现了通信双方之间一般意义下的时序上的约束。在并发面向对象模型中，同步控制往往是指：当向一个共享对象发送消息时，消息接受者不是无条件地处理发送来的消息，它必须根据自身目前的状态来决定能处理哪些消息，对不能处理的消息可以让它等待；当有多个消息时，对象还要控制哪些消息的处理可以并发执行，即，对象必须对消息处理进行同步。另外，如果对象有自己的私有线程，那么，对象还要对私有线程和消息处理线程的执行进行同步控制。

3.3.1 条件同步与互斥

在论文的并发面向对象模型中，把一个对象的同步控制分成两类：条件同步和互斥。对象的条件同步控制是指对象在某种状态下只能接收某一些消息，如：对于有界缓存(Bounded Buffer)对象：当缓存为空时，只能接收put消息而不能接收get消息；缓存满时，只能接收get消息而不能接收put消息。对象的互斥控制是指：当处于某种状态下的对象可以接收(处理)多个消息，如：缓存不空又不满，这时，既能接收get消息又能接收put消息，由于这些消息的处理都有可能使用对象的局部变量，如果让它们同时执行，就存在共享变量(对象的局部变量)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对象的局部变量，否则将造成对象状态的混乱。

条件同步控制用于控制对象的消息接收，一旦对象接收了某个满足条件同步限制的消息后，该消息的处理中就不再受条件同步的控制了，即，消息处理在执行过程中不会再等待其它条件的产生，如果需要等待也只是为了互斥使用对象的局部变量；互斥控制用于控制对象内部执行线程的并发操作，以保证对象状态的完整性。

3.3.2 缺省并发控制

在本论文的并发对象模型中，对象的并发控制采用了分散控制方式，即，把消息的接收控制和消息处理的互斥控制分散到对象的各方法上。

3.3.2.1 条件同步的描述

为了描述对象的条件同步，在定义并发对象类时，给每个方法设置一个激励条件，它是一个布尔表达式，其中的运算分量可以是对象的局部变量和相应方法的形式参数。由于对象的一些并发状态(如：对象的操作历史)用局部变量表示起来很麻烦，而这些状态往往对于实现对象的消息调度控制是很有用的，因此，为了方便地表示基于操作历史的条件同步，论文设计了一些历史函数，例如：

- is_running(<方法>)。
  true: <方法>正在执行
  false: <方法>不在执行

- is_waiting(<方法>)。
  true: <方法>对应消息正在等待执行
  false: <方法>对应消息不在等待执行

- waiting_before(<方法1>,<方法2>)
  true: 存在等待处理的与<方法1>和<方法2>所对应的消息，并且
  <方法1>对应消息先于<方法2>的消息到达。
  false: 不存在上述情形。

这些历史函数也可以出现在方法激励条件的布尔表达式中。在论文的并发面

向对象模型中，历史函数是作为对象的局部方法来实现的，并发对象类的设计者可以根据需要自行设计历史函数，这体现了系统的可扩充性。由于历史函数只获取对象的状态而不改变之，因此，激励条件中的方法调用不受对象并发控制的限制。

对象的条件同步控制代码由实现系统根据方法的激励条件自动产生，当对象接收消息时，将隐式地调用这段代码。

3.3.2.2 顺序的条件同步

在本文所提并发面向对象模型中，对象的缺省并发控制是顺序的条件同步。对象接收消息的先决条件是相应方法的激励条件成立，对不满足条件的消息让其在对象外等待。当对象接收了一个消息后，自动变成关闭状态，使得对象不再接收其它消息，以保证消息处理的互斥性。消息处理过程执行结束后，自动开放对象，使得对象可以接收下一个消息的处理。其中，对象处于开放状态是对象接收消息的一个隐含条件。当某消息所对应的方法没有显式的激励条件时，只要对象处于开放状态，就能接收该消息进行处理。

如果对象在处理某消息的过程中又向自己同步发送了一个消息，即，调用了本对象的其它方法，则该消息处理不受对象条件同步的控制，它相当于一般的过程调用，这样可避免直接递归消息传递所引起的死锁问题。

当在某消息的处理过程中创建了对象的一个私有线程，这时相当于对象向自己异步发送了一个消息。对象的私有线程没有显式的激励条件，对象处于开放状态是私有线程执行的唯一条件，私有线程的执行也要受对象条件同步的控制。

这里的做法与一些基于管程的并发系统不同，管程的实现保证了其内部操作的互斥性，但条件同步控制是在内部操作过程中通过条件变量上的wait和signal操作来完成的，这样，将使得条件同步控制变得很复杂，缺少系统性，并且也不利于并发对象类的继承。

3.3.3 内部并发控制

论文中之所以要求对象在接收一个消息后自动关闭，不再接收其它消息的处理，除了是为了保证互斥外，还有另一个考虑，就是复用已有的顺序面向对象模型中的对象类，使得这些顺序对象类在并发环境中以自然的互斥语义出现，从而保证以这些类所创建的对象的状态不受并发操作的破坏。

对象的条件同步控制实现了对象之间的并发控制。在论文所提的并发面向对

象模型中，对象内部操作之间还可以存在并发，为了实现对象内部操作的并发控制，在本论文的并发面向对象模型中，允许在消息处理过程和私有线程中适当的时候显式地开放对象，使得其它满足激励条件的消息处理或私有线程得以执行，这时，开放对象的消息处理过程和私有线程的实现中应不得再使用对象的局部变量，以保证互斥。如果消息处理过程或私有线程在显式地开放对象一段时间后又需使用对象的局部变量，则在其实现中应保证在使用前首先显式地关闭对象，若这时对象已被其它线程关闭，则必须等待关闭对象的线程开放对象。对象关闭后，就不再接收新的消息，而正在执行的消息处理线程和私有线程照常执行，除非它们也在执行关闭对象操作(准备使用对象的局部变量)。

若对象有私有线程，而私有线程的实现中存在无限循环，则应在适当的时候显式地开放对象，使得对象能接受其它消息进行处理(具体如何控制参见实例3)。

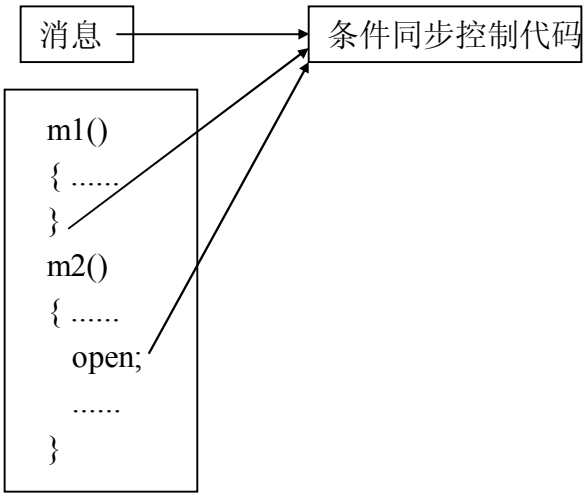
为了避免间接递归消息传递可能产生的死锁，当在消息处理线程或私有线程中向其它对象发送消息时，可以在消息发送前先显式地开放本对象，使得对象在等待其它对象处理消息时仍然可以接收消息。为了减小对象内部并发控制的复杂性，可以采用一些并发的局部对象来实现对象局部变量的管理。

在本论文的并发面向对象模型中，对象的条件同步控制代码在下述几种情况下将被自动调用：

(1)有新消息到达。

(2)一个消息处理线程或私有线程执行结束。

(3)在一个消息处理线程或私有线程执行中显式地开放对象。



3.4 并发对象类的继承

在论文的并发面向对象模型中，并发对象类的继承是通过把类的并发描述和功能描述(方法的实现)分开进行实现的。并发对象类方法的继承与顺序面向对象模型的做法相同。下面着重介绍并发对象类中有关并发行为的继承。

3.4.1 条件同步控制的继承

在定义一个并发对象类的子类时，其方法的激励条件可以从父类继承，继承机制定义如下：

(1)如果在一个并发对象类的子类中，重定义了父类的一个方法，但没有定义其激励条件，这时，该方法继承父类中相应方法的激励条件；

(2)如果在一个并发对象类中，重定义了父类的一个方法的激励条件，则父类方法的激励条件对子类不再有效。

(3)如果在子类中对父类的激励条件进行重定义时，需要用到父类的激励条件，则必须在子类激励条件的布尔表达式中显式地把它表示出来，即，把父类的激励条件作为子表达式加入到子类激励条件的布尔表达式中。

上述(3)的做法与其它采用激励条件实现对象并发控制的系统有所不同，在这些系统中，激励条件是隐式继承的，子类方法的激励条件是由子类中定义的条件与父类相应方法激励条件的‘或’运算构成[56]。与这些系统相比，本论文的做法具有更强的表达能力。

3.4.2 内部并发控制的继承

在论文的并发面向对象模型中，对象的内部并发控制是在对象的方法中实现的，因此，在定义子类时，其内部并发控制随方法的实现一起从父类继承。

在论文第二章中提到：把对象的并发控制放在对象的方法中实现会造成继承异常[59]，在本论文所提出的并发面向对象模型中，之所以没有把对象的内部并发控制从方法的实现中分离出来，主要出于下面两个方面的考虑：

(1)利弊的权衡。

如果把对象的并发控制(包括条件同步控制和内部并发控制)完全放在对象方法中实现，则具有较强的表达能力，但会使得对象的并发控制缺乏系统性，同时也不利于并发对象类的继承；如果把对象的并发控制完全从对象方法的实现中分离出来，则对象并发控制描述比较清晰，但表达能力下降。因此，论文采用了一种折衷的方案，即，把对象的条件同步控制从对象方法的实现中分离出来，用激励条件来实现，而对象的内部并发控制仍然放在对象的方法中实现，这样可以发挥各自的长处。

(2)继承异常程度

虽然把并发控制放在对象方法中实现易造成继承异常，但对于本论文所提并发面向对象模型的并发控制来说不是那么严重。因为，在本模型中，对象内部的并发控制主要是为了实现对象内部各线程互斥地使用对象的局部变量。对象的内部并发控制往往要涉及到对象方法实现中是否使用对象的局部变量，即，对象的内部并发控制与对象方法功能的具体实现有关。在定义子类时需要修改父类的内部并发控制，往往是由于对父类中象某方法的实现进行重定义所造成的，如：在子类中把父类的某方法重定义成不再使用对象的局部变量或又使用了新的局部变量。因此，论文认为，由某方法实现的修改而引起的对象内部并发控制的修改，造成对父类其它方法进行重定义的范围较小，因为其它方法对对象局部变量的使用方式没有变，它们原来的互斥控制仍然有效，只是重定义的那个方法的互斥控制需要修改。

3.5 实例

为了进一步说明本论文所提出的并发面向对象模型的一些基本思想，下面给出一些采用该模型来描述的一些典型的并发系统实例，实例的描述中使用了类C++语法。

例 1、有界缓存(bounded buffer)

有界缓存对象由一个共享资源：buffer和两个消息处理方法：put与get所组成。其典型的实现可以用下面的BoundedBufl类来描述：

```
class BoundedBufl
```

```
{ Product buffer[100];
  int in,out,max_len;
  BoundedBuf1()
  { in = out = 0;  max_len = 100;
  };
  Product get(): in != out
  { Product i=buffer[out];
    out = (out+1)%max_len;
    return i;
  };
  void put(Product i): (in+1)%max_len != out
  { buffer[in] = i;
    in = (in+1)%max_len;
  }
}
```

方法get与put分别有一个激励条件，用于控制get和put消息的接收。在这个实现中，对象内部不存在并发，消息处理顺序执行。如果允许get与put并发执行(因为它们分别在buffer的两端进行操作)，则相应的实现可表示成：

```
class BoundedBuf2
{  ......
  Product get(): in != out && !is_running(get)
  { open(); // 开放对象
    Product i=buffer[out];
    out = (out+1)%max_len;
    return i;
  };
  void put(Product i): (in+1)%max_len != out && !is_running(put)
  { open(); //开放对象
    buffer[in] = i;
    in = (in+1)%max_len;
  }
```

```
}
```

如果允许多个get和一个put操作，即典型的多读者/单写者(readers/writer)应用，则有界缓存的定义可改写成：

```
class BoundedBuf3
{ ......
  Product get(): !is_running(put)
  { open(); // 开放对象
    Product i;
    读取 i;
    return i;
  };
  void put(Product i): !is_running(get) && !is_running(put)
  { 存储 i;
  }
}
```

上述定义中存在“饿死”现象，即，由于get消息不断到达，使得put消息永远得不到处理，对于这个问题，可采用下述解决办法：

```
class BoundedBuf4
{ ......
  Product get(): !is_running(put) && !waiting_before(put,get)
  { open(); // 开放对象
    Product i;
    读取 i;
    return i;
  };
  void put(Product i): !is_running(get) && !is_running(put) && !waiting_before(get,put)
  { 存储 i;
  }
}
```

如果采用继承机制，则BoundedBuf4也可以定义成BoundedBuf3的子类，其中，只对方法get与put的激励条件进行了重定义，方法的实现从BoundedBuf3继承：

```
class BoundedBuf4: public BoundedBuf3
{  Product get(): BoundedBuf3::activation && !waiting_before(put,get);
   void put(Product i): BoundedBuf3::activation && !waiting_before(get,put);
}
```

其中，BoundedBuf3::activation表示引用父类相应方法的激励条件。

例 2、生产者与消费者问题。

在这个问题中有三个对象：生产者、消费者及缓存。其中生产者与消费者有初始私有线程，一旦被创建，它们的初始私有线程立即开始执行。缓存是一个资源管理对象，它提供了两个消息处理方法put与get，它们分别加上了用于条件同步的激励条件。在这个例子中，对象内部没有并发。下面给出了生产者对象类Producer的定义、消费者对象类Consumer的定义以及缓存、生产者和消费者对象的创建。

```
class Producer
{ BoundedBuf *p_buf;
  Product item;
  Producer(BoundedBuf *p) { p_buf = p; };
  void init_thread() //初始私有线程
  { while (1)
    { 生产物品 item;
      p_buf->put(item); //把 item 放入缓存
    }
  }
}

class Consumer
{ BoundedBuf *p_buf;
  Product item;
```

```
  Consumer(BoundedBuf *p) { p_buf = p; };
  void init_thread() //初始私有线程
  { while (1)
    { item = p_buf->get(); //从缓存取 item
      消费物品 item;
    }
  }
}

BoundedBuf buf;
Producer p(&buf);
Consumer c(&buf);
```

例 3、哲学家就餐问题。

在这个问题中有六个对象：五个哲学家和一个餐桌。哲学家除了有一个初始私有线程外，还有一个让他人提交问题的方法new_problem，该方法没有显式的激励条件，其隐含的激励条件是哲学家对象处于开放状态。在哲学家的初始私有线程中，为了使得他人有机会提交问题，在准备就餐前和无思考问题时，调用open操作开放对象。在思考问题时，哲学家对象是关闭的，因为有可能要使用局部变量。当然，也可视思考问题的具体情况，在思考问题的实现中关闭/开放对象。餐桌对象管理着五个叉子并提供了一个方法eat，eat的显式激励条件是：就餐的某个哲学家的左右叉子没被占用。得到叉子的哲学家在就餐前调用open操作开放餐桌对象，使得其它哲学家有机会就餐。在这个例子中，对象内部存在并发。下面给出了哲学家对象类的定义、餐桌对象类的定义以及餐桌和五个哲学家对象的创建。

```
class Philosopher
{ int no;
  Table *p_table;
  Philosopher(int i, Table *t)
  { no = i; p_table = t; };
  void new_problem(...) {...};
  void init_thread()
```

```
  { while (1)
    { if 有思考问题
      { 思考问题;
        open(); //让他人提交新问题
        p_table->eat(no); //准备就餐
      }
      else
        open(); //让他人提交问题
      close(); //准备思考下一个问题
    }
  }
}

class Table
{ BOOL forks[5];
  Table()
  { for (int i=0; i<5; i++) forks[i] = TRUE;
  };
  void eat(int no): forks[no]&&forks[(no+1)%5]
  { forks[no] = FALSE;
    forks[(no+1)%5] = FALSE;
    open(); //让其它哲学家就餐
    就餐;
    close(); //修改局部变量 forks
    forks[no] = TRUE;
    forks[(no+1)%5] = TRUE;
  }
}

Table table;
Philosopher ph0(0,&table),ph1(1,&table),ph2(2,&table),ph3(3,&table),ph4(4,&table);
```

3.6 与相关工作的比较

为了更加清楚地说明本章所提并发面向对象模型的特点，现从下面几个方面将它与现有的一些并发面向对象模型作一比较：

1. 并发与面向对象模型的集成

论文从面向对象模型的特点出发，以一种集成的方式把并发引进到面向对象模型中来，特别是，论文所提模型中不区分主动对象与被动对象，模型中只有一种对象：并发对象，它们是自治的并发单位，都具有并发控制能力，可以用于各种环境。这与一些基于主动对象的并发面向对象系统[10,19,50]有很大的不同，在这些系统中，除了对象以外，还有一个进程概念，虽然它们也用(主动)对象来称呼进程，但对于使用者来说，确实存在两种概念，他们必须知道所使用的是一般对象(被动对象)还是进程(主动对象)，这将给软件的设计、理解和重用带来不便。

2. 系统的并发执行

本模型的并发执行是通过对象的私有执行线程来实现的，对象间采用同步消息发送(远程过程调用)。这个做法与一些采用异步消息传递的系统(如：actor模型)不同。在采用异步消息传递的系统中，系统的并发执行是由异步消息发送产生，而本模型的并发执行是通过对象的私有线程产生。虽然从效果上将，私有线程的执行也是一种异步形式，但从对象并发控制的角度来看，本模型的并发结构属于同步接收、异步执行，它使得程序设计者对程序的并发行为可以有一定的控制。

本模型与其它一些采用私有执行线程的并发面向对象系统(如：POOL和Eiffel//)也存在不同，在这些系统中，私有线程(对象体)的作用是用于对象的并发控制，而在本模型中，私有线程是用于对象内部的事务处理，并发控制由其它方式实现。用对象体实现对象的并发控制会引起继承异常。

虽然，本模型与CEiffel的做法有些相似，但也存在差异。在CEiffel中，对象的私有线程都是在对象创建时自动执行，一个线程执行结束后又自动重新执行；而在本模型中，除了初始私有线程外，其它私有线程由对象的方法来启动，这样做的好处在于：避免对象在没有内部事务需要处理时不停地启动私有线程，因为对象内部的事务处理一般是在接收了一个消息后产生的。另外，在对象内部并发控制和继承方面，本模型与CEiffel也存在不同(详见3)。

3. 对象的并发控制

在本模型中，对象之间和对象内部都存在并发，对象的并发控制采用了分散的隐式与显式相结合的控制方式，即，对象的条件同步控制由方法的激励条件实现，而对象的内部并发控制则在对象方法中实现。

大多数现有的并发面向对象模型不允许对象内部的并发，在这些系统中，对象顺序地从消息队列中选取满足执行条件的消息进行处理，这样，不仅限制了系统的表达能力，而且递归的消息发送易造成死锁(不管采用的是同步还是异步消息传递)。

在允许对象内部并发的并发面向对象模型(如:CEiffel)中，内部并发控制往往放在方法执行前进行控制，一旦满足执行条件，方法的执行中就不再受对象的内部并发控制的限制，可以任意使用对象的局部变量，这样做的好处是便于对象类的继承，不足之处在于限制了系统的表达能力。

与其它基于激励条件的系统(如：DRAGOON和CEiffel等)相比，本模型也有不同。首先，本模型的激励条件是实现对象的条件同步控制，而其它模型的激励条件既用于实现条件同步控制，又用于实现对象内部并发控制；其次，在其它模型中，激励条件是隐式继承，子类的激励条件与父类的激励条件是‘或’的关系，这除了表达能力问题以外，还有一个多继承问题，即，从哪一个父类继承同名方法的激励条件？最后，在其它模型的激励条件中描述消息接收调度的能力受到限制，如DRAGOON中只能用有限的几个历史函数来实现，而本模型中，激励条件中允许有局部方法调用，这样，对象类的设计者可以自行定义调度策略。

4. 对继承机制的支持

大多数并发面向对象模型在引进并发描述时，着重考虑的是系统的表达能力，所采取的并发机制往往会破坏面向对象的继承机制，造成继承异常。关于继承异常的研究普遍认为：解决继承异常的办法是把对象的并发控制描述与对象的功能性描述分开实现，但要做到这一点往往很困难，因为，如果把这两者完全分开，则系统的表达能力将受到影响。本论文采取了一种折衷的解决方案，把对继承影响较大的条件同步控制从方法实现中分离出来，从而减轻了继承异常的程度，同时又兼顾了系统的表达能力。

3.7 小结

本章从对象系统的并发执行、对象间的通信、对象的并发控制和并发对象类的继承四个方面介绍了论文提出的基于并发对象的并发面向对象模型。该模型强调并发与面向对象模型的集成。模型采用私有线程机制在面向对象模型引进并发执行，对象间采用远程过程调用方式进行通信，对象的并发控制由条件同步控制和内部并发控制两种，条件同步控制用方法的激励条件来描述，内部并发控制由对象的方法实现。对继承的支持是：条件同步控制可以与方法的实现分开继承。本章还用一些典型的实例对所提并发面向对象模型进行了进一步的描述，最后，把所提模型与现有的一些并发面向对象模型进行了比较。

第四章 并发对象的形式语义

为了对论文第三章所提出的并发面向对象模型中的并发行为有较精确的描述，本章将给出该模型中并发对象的形式语义。由于目前对CSP[43]语义的研究相对来说已比较成熟[94]，因此，本章采用CSP来描述论文所提并发面向对象模型的并发对象的语义。本章首先对CSP作一简单介绍，然后，用CSP来分别描述论文所提的并发面向对象模型中对象间消息传递、对象缺省条件同步控制以及对象内部并发控制的语义。

4.1 CSP 简介

CSP(Communicating Sequential Processes)是C. A. R. Hoare设计的基于同步消息传递和选择式通信的一种并发程序设计语言[43]，虽然它主要是为研究目的而设计的，没有确定的语言定义版本，但其中的很多思想对后来的一些并发程序设计语言产生了巨大的影响[47,75]。下面对论文中将要用到的CSP成分作简单介绍。

CSP是一个命令式语言，一个CSP程序就是一个进程，每个进程可以通过并发进程运算符||分解为若干个并发执行的子进程，这种分解可以嵌套至任意深度。

在CSP中，进程间通信采用了一对输入/输出原语：“<源进程名>?<变量>”和“<目标进程名>!<表达式>”，只有当通信的两个进程中一个执行到输入语句，而另一个执行到输出语句，并且各自指定对方为输入/输出对象，通信才能进行，如果有一方先到达输入/输出语句而另一方还未到达相应的输出/输入语句，则先到达的必须等待(同步消息传递)。通信建立后，输出语句中的表达式传给输入语句中的变量，这是进程间交换数据的唯一方式。一个进程不能修改另一个进程的变量。进程的创建是静态的，进程间通信采用显式的通信对象命名规则，进程不能作为变量的值。

CSP中引进了不确定的选择语句，其中的选择项是带卫士(guard)的语句：

[$<guard_1> \rightarrow <statement\_list_1>$

$<guard_2> \rightarrow <statement\_list_2>$

.....

<$guard_n$> → <$statement\_list_n$>
]

其中，<$guard_i$>可以是:
(1)<布尔表达式>
(2)<输入语句>
(3)<布尔表达式>;<输入语句>

选择语句将不确定地选择一个满足<$guard_i$>的<$statement\_list_i$>执行。若所有的<$guard_i$>均不满足，则选择语句失败。

CSP还提供了重复语句：

*<选择语句>

它重复地执行<选择语句>，直到<选择语句>的所有<$guard_i$>均不满足。

4.2 并发对象的定义

为了描述论文第三章所提并发面向对象模型中的并发对象的语义，这里给出简化了的并发对象的定义：

(1)系统由一组静态创建的并发对象O_1，O_2，...，O_n构成；
(2)对象间采用远程过程调用形式进行消息传递：$O_i.m_j(...)$；
(3)每个对象O_i由一组局部变量$(v_1,v_2,...,v_r)$、一组方法$(m_1,m_2,...,m_k)$和一组私有线程$(t_1,t_2,...,t_s)$构成，
(4)每个方法m_j由激励条件$cond_j$、形参表$param_list_j$、返回值和方法语句表$m_st_list_j$构成;
(5)每个私有线程t_j由线程语句表$t_st_list_j$构成；
(4)方法语句表m_st_list和线程语句表t_st_list由一般语句、远程过程调用语句$O_i.m_j(...)$、局部方法调用$m_j(...)$和私有线程调用$t_j()$构成；
(5)对象O_i可以有一个名为$init_thread_i$的初始私有线程，它有初始语句表

$init_st_list_i$构成。

下面将给出上述模型的CSP语义。

4.3 并发对象的 CSP 语义

进程构成了CSP程序并发执行的单位，在用CSP描述论文所提并发面向对象模型中的并发对象时，论文首先考虑如何把并发对象表示成进程，其次对这些进程的行为进行描述，最后给出对象间消息传递的CSP描述。

4.3.1 对象的进程模型

一个对象O_i可表示成下面五个CSP进程：

(1)LocalVarManager(i)
实现对象O_i的局部变量管理。

(2)Mutex(i)
实现对象O_i的互斥控制。

(3)InitThread(i)
实现对象O_i的初始私有线程。

(4)Method(i,j) (j=1..k_i, k_i为对象O_i方法的个数)
实现对象O_i的方法。

(5)Thread(i,j) (j=1..s_i, s_i为对象O_i私有线程的个数)
实现对象O_i的私有线程。

整个并发面向对象系统由所有对象的上述进程通过并发运算符构成，它们并发执行：

SYSTEM ＝ [LocalVarManager(i) || Mutex(i) || InitThread(i) ||
Method(i,1) || Method(i,2) || ... || Method(i,k_i) ||
Thread(i,1) || Thread(i,2) || ... || Thread(i,s_i)]　(i=1..n)

下面将分别给出这些进程的CSP描述

4.3.2 对象各进程的 CSP 表示

在下述的描述中，有些地方采用了非CSP表示法，这仅仅是为了方便起见，它们完全可以用CSP来表示。

1. LocalVarManager(i)

LocalVarManager(i) :: declaration $v_1,v_2,...,v_r$;

*[(j:1..k_i) Method(i,j)?ReadAll→Method(i,j)!($v_1,v_2,...,v_r$)

(j:1..k_i) Method(i,j)?WriteAll→Method(i,j)?($v_1,v_2,...,v_r$)

(j:1..k_i) Method(i,j)?Read(h)→Method(i,j)!(v_h)

(j:1..k_i) Method(i,j)?Write(h)→Method(i,j)?(v_h)

]

在对象的Method、Thread和InitTread进程中若要使用对象的局部变量，则可通过向LocalVarManager发送消息来实现。

2. Mutex(i)

Mutex(i) :: val:integer; val:=1;

*[(j:1..k_i) Method(i,j)?unlock() → val := 1

val = 1; (j:1..k_i) Method(i,j)?lock() → val := 0

]

3. InitThread(i)

InitThread(i):: Mutex(i)!lock(); init_st_list; Mutex(i)!unlock()

4. Method(i,j)

Method(i,j) :: *[(u:1..n,v:1..k_u,u≠i)Method(u,v)?m_j(param_list$_j$)→
Mutex(i)!lock();
*[cond$_j$＝FALSE→Mutex(i)!unlock(); Mutex(i)!lock()];
statement_list$_j$;
Mutex(i)!unlock();
Method(u,v)!result 或 Method(u,v)!m_j()

(v:1..k_i, v≠j)Method(i,v)?m_j(param_list$_j$)→
statement_list$_j$;
Method(i,v)!result 或 Method(i,v)!m_j()

(u:1..n,v:1..s_u, u≠i)Thread(u,v)?m_j(param_list$_j$)→
Mutex(i)!lock();
*[cond$_j$＝FALSE→Mutex(i)!unlock(); Mutex(i)!lock()];
statement_list$_j$;
Mutex(i)!unlock();
Thread(u,v)!result 或 Thread(u,v)!m_j()

(v:1..s_u)Thread(i,v)?m_j(param_list$_j$)→
statement_list$_j$;
Thread(i,v)!result 或 Thread(i,v)!m_j()

(u:1..n, u≠i)InitThread(u)?m_j(param_list$_j$)→
Mutex(i)!lock();
*[cond$_j$＝FALSE→Mutex(i)!unlock(); Mutex(i)!lock()];
statement_list$_j$;
Mutex(i)!unlock();
InitThread(u)!result 或 InitThread(u)!m_j()

InitThread(i)?m_j(param_list$_j$)→
statement_list$_j$;
InitThread(i)!result 或 InitThread(i)!m_j()

]

在上述的描述中，“或”表示根据方法是否有返回值来决定。

上述描述的直观含义是：从其它对象的方法、私有线程和初始线程发送来的消息要受对象并发控制的限制，而从本对象的方法、私有线程和初始线程发送来的消息不受对象并发控制的限制。

在第三章的并发面向对象模型中，允许并发调用对象的同一个方法，上面的描述没有给出对同一个Method(i,j)并发调用的描述，解决办法是采用进程数组[43]。

5. Thread(i,j)

Thread(i,j) :: *[(v:1..k_i)Method(i,v)?t_j()→
Method(i,v)!t_j();
Mutex(i)!lock();
t_st_list$_j$;
Mutex(i)!unlock();

(v:1..k_i)Thread(i,v)?t_j()→
Thread(i,v)!t_j();
Mutex(i)!lock();
t_st_list$_j$;
Mutex(i)!unlock();

InitThread(i)?t_j()→
InitThread(i)!t_j();
Mutex(i)!lock();
t_st_list$_j$;
Mutex(i)!unlock();
]

4.3.3 消息传递语义

1. 远程过程调用

在一个对象的方法或私有线程中向其它对象发送消息，根据消息是否有返回

值，可分别表示成下面的CSP形式：

(1)无返回值

$O_i.m_j(exp\_list_j)$ ==> $Method(i,j)!m_j(exp\_list_j)$; $Method(i,j)?m_j()$

(2)有返回值

$x = O_i.m_j(exp\_list_j)$ ==> $Method(i,j)!m_j(exp\_list_j)$; $Method(i,j)?x$

2. 局部方法调用

与1相同

3. 私有线程调用

在一个对象O_i的方法或私有线程中启动私有线程$t_j()$，可表示成下面的CSP形式：

$Thread(i,j)!t_j()$; $Thread(i,j)?t_j()$

4.4 小结

本章用CSP对第三章所提并发面向对象模型中的并发对象的行为进行了描述，在描述中，把一个对象看成是由若干个并发执行的进程构成，整个系统由所有对象的这些进程通过并发运算符得到。

第五章 NDC++ : C++的并发扩充

对于一种并发面向对象模型，如何用程序设计语言来描述它，一般存在下述三种途径[50]：

(1)设计一种新的支持并发的面向对象程序设计语言(COOP)[10,72,92]。
(2)在某种顺序面向对象程序设计语言中引进并发成分(扩充)[19,91,49]。
(3)提供并发类库[50]。

显然，(1)是一种比较理想的途径，但是所需花费的代价太大，并且，一个新语言要得到人们的认可，需要经过相当长的一段时间。(3)是一种最省事的途径，但仅有类库是不够的，还必须有相应语言成分的支持，否则一些新的思想难以彻底体现。因此，我们采用了把(2)与(3)相结合的途径，即，在某种顺序面向对象程序设计语言(称为基语言)中引进一些用于描述并发的成分，用扩充后的语言加上所提供的并发类库进行并发面向对象程序设计，对得到的结果，利用一个预先设计好的转换程序把其转换成基语言程序，然后用基语言编译程序把它编译成可执行代码。

这里，我们选择C++作为基语言，是因为它是一种较为流行并且使用较广的面向对象程序设计语言。在论文中，扩充的C++语言被称为NDC++。

为了能描述并发计算，NDC++在C++的基础上，引进了一些并发描述和控制设施，其中包括：一个预定义的类**Concurrency**和在C++的类定义中增加了两个部分：**concurrency_control**和**threads**，另外，还引进了一个关键词：**activation**。下面分别对这些成分进行介绍。

5.1 预定义的 Concurrency 类

为了能用C++描述第三章中的并发面向对象模型，在NDC++中首先设计了一个类：Concurrency，其定义如下所示。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();
    void new_thread(FUNC *pThread);
    void stop_threads();
    virtual void init_thread();
    int is_waiting(char* pMethodName);
    int is_running(char* pMethodName);
    BOOL waiting_before(char* pMethodName1, char* pMethodName2);
    int register_waiting(char* pMethodName);
    int register_running(int index);
    void register_finished(int index);
}
```

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，该成员函数等待其它成员函数开放(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)**init_thread**()用于定义对象的初始私有线程，Concurrency类中的**init_thread**()为虚函数，它不做任何事情，子类中可以对其重定义。

(4)new_thread()用于创建一个新线程以执行参数所指定的函数。

(5)stop_threads()用于终止对象所有私有线程的执行。

(6)is_waiting()用于判断是否有参数所指定的消息在等待处理，返回值为0表示没有，否则返回等待消息的个数。

(7)is_running()用于判断是否有参数所指定的消息处理(成员函数)在执行，返回值为0表示没有，否则返回在执行的消息处理的个数。

(8)waiting_before()用于判断等消息待队列中是否有参数1所指定的消息在参数2所指定的消息之前到达，返回值为0表示没有，否则表示存在。

(9)register_waiting()用于实现等待消息的队列管理，把参数所指定的消息放入等待消息队列，返回值为等待消息队列的索引。

(10)register_running()用于实现执行(正在处理的)消息的队列管理，把参数所指定的消息从等待消息队列中删除，放入执行消息队列中，返回值为执行消息队列的索引。

(11)register_finished()用于消息处理结束后，把它从执行消息队列中删除。

Concurrency类所提供的函数可以用来实现其子类对象的并发控制。当对象接受到一个消息时，将采用下面的次序来使用Concurrency类中的函数：

(1)调用register_waiting()把它放入等待消息队列；

(2)调用lock()关闭对象，如果成功，则继续步骤(3)，否则，等待；

(3)判相应成员函数的激励条件，若条件满足，则继续步骤(4)，否则，调用unlock()开放对象，重复步骤(2)；

(4)调用register_running()把消息从等待消息队列中删除并放入执行消息队列中；

(5)成员函数开始执行直到结束；

(6)调用register_finished()把消息从执行消息队列中删除

(7)调用unlock()开放对象，消息处理返回。

为了实现对象内部的并发，在成员函数的执行中，可调用unlock()释放对象，使得其它被锁住的消息就能得到处理。若显式地调用了unlock()，则在相应成员函数中应保证在下一次调用lock()前不对成员变量进行操作。

另外，在定义Concurrency类的子类成员函数的激励条件时，可把对is_waiting()、is_running()和waiting_before()函数的调用作为条件项，以描述消息接收的一些调度策略。如果这些历史函数不能实现所需的消息调度策略，则在子类中可自行定义所需要的消息调度函数，这些函数按照继承机制可以出现在子类、子类的子类、...的激励条件中。

5.2 语言扩充

上述的Concurrency类是作为C++类库中的一个类来实现的。如果在设计一个并发对象类时直接调用Concurrency类中的函数来实现对象的并发控制，这将增加设

计者的负担，设计者必须要知道这些函数的调用协议，即，何时以及采用何种次序来调用它们，使用不当将达不到预期的效果，甚至引起错误。

为了能让设计者以一种较抽象的形式来描述对象的并发控制，NDC++在C++的基础上，引进了一些用于描述对象并发控制的成分，采用这些成分来描述对象的并发控制，将大大减轻设计者的工作量，并使得设计出的并发对象类更具有结构性。下面分别对这些成分进行介绍。

5.2.1 条件同步定义 concurrency_control

为了描述对象的条件同步控制，我们在C++类定义中原有成分之后增加了一个新的成分：**concurrency_control**，它用于定义成员函数的激励条件，其中的每一项具有下述形式：

<成员函数首部>：[<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：

(1)常量
(2)对象的成员变量；
(3)相应成员函数的形式参数；
(4)对本对象其它成员函数的调用；以及
(5)父类中相应成员函数的激励条件，用<父类名>::**activation来**表示。

如果<激励条件>缺省，则表示不继承父类相应成员函数的激励条件(详见5.2.3)。下面给出了一个引进**concurrency_control**之后的C++类定义模板：

```
class concurrent_obj: public Concurrency
{ public:
    ret_type_1 func_1(param_list_1) { ..... };
        :
    ret_type_n func_n(param_list_n) { ..... };
 concurreny_control:
    ret_type_1 func_1(param_list_1): condition_1;
        :
```

```
    ret_type_n func_n(param_list_n): condition_n;
}
```

激励条件的作用是用于控制对象消息的接收，即，实现对象的条件同步控制，相应控制代码由实现(5.3的转换机制)产生。对象内部的部分控制则由并发类的设计者下调用Concurrency类中的lock()和unlock()来实现。下面给出了可以在缓存两端分别进行读写的并发对象类BoundedBuf的定义。

```
class BoundedBuf: public Concurrency
{ protected:
    Product buffer[100];
    int in,out,max_len;
  public:
    BoundedBuf()
    { in = out = 0;  max_len = 100;
    };
    Product get()
    { unlock();
      Product i=buffer[out];
      out = (out+1)%max_len;
      return i;
    };
    void put(Product i)
    { unlock();
      buffer[in] = i;
      in = (in+1)%max_len;
    };
  concurrency_control:
    void put(Product i): (in+1)%max_len != out && !is_running(put);
    Product get(): in != out && !is_running(get);
}
```

5.2.2 私有线程定义 threads

为了描述对象的私有执行线程，NDC++在C++的类定义中引进了另一个新的成分：**threads**，它位于**concurrency_control**之后。**threads**部分由一些成员函数的定义构成，其中可以有一个名字为**init_thread**的成员函数定义，这些成员函数表示私有线程，**init_thread**为初始私有线程。私有线程定义在形式上与其它成员函数定义基本相同，差别在于：

(1)私有线程没有形式参数和返回值；
(2)私有线程只能由对象的方法来调用；
(3)私有线程一旦被调用，它们将与调用者并发执行；
(4)如果**init_thread**被定义，则在对象创建时，它将自动执行。

下图给出了一个引进**threads**之后的C++类定义模板：

```
class concurrent_obj: public Concurrency
{ .......;
  concurreny_control:
    .......;
  threads:
    void thread_1() { ..... };
    .......;
    void thread_n() { ..... };
    [ void init_thread(); ]
}
```

对私有线程的调用相当于对象向自己发送了一条消息，私有线程的执行也要受对象并发控制的限制。私有线程没有显式的激励条件，其隐含的执行条件是：对象处于开放状态。

5.2.3 并发对象类的继承原则

在NDC++中定义并发对象类的子类时，其成员函数的继承原则与C++相同，这里不再详述。下面着重叙述NDC++中成员函数激励条件的继承机制。

在定义并发对象类的子类时，其成员函数的激励条件可以从父类继承，继承原则如下：

(1)在子类中重定义了父类的一个成员函数但没有定义其激励条件，则该成员函数的激励条件继承父类中相应成员函数的激励条件。

(2)在子类中如果重定义了父类一个成员函数的激励条件，则该激励条件是完全的重新定义，父类相应成员函数的激励条件不作为其一个隐含的‘与’或‘或’条件项，如果重定义的激励条件中需要父类的激励条件作为其一个条件项，则必须用：<父类名>::**activation** 显式地给出。

(3)在子类中如果需要取消父类某成员函数的显式激励条件，则必须在子类中的**concurrency_control**部分把该成员函数的激励条件定义为空。

(4)在多继承情况下，必须显式地指出子类成员函数的激励条件从哪一个父类的成员函数的激励条件继承。

5.3 转换机制

为了使得扩充的并发对象类定义能够被C++编译程序识别，我们设计了一个转换程序，用以把NDC++的类定义转换成C++的形式，其转换策略如下：

5.3.1 成员函数的转换

对类中的每个成员函数m，首先把m改名为_m，然后根据类中是否给出了m的激励条件分别对m进行重新定义：

(1)m有激励条件B。

(a)m无返回值

```
void m(...)
{ int _index=register_waiting("m(...)");
  lock();
  while (!B) { unlock();  lock();}
  register_running(_index);
  _m(...);
  register_finished(_index);
  unlock();
```

```
};
```

(b)m有返回值

```
ret_type m(...)
{ int _index=register_waiting();
  lock();
  while (!B) { unlock(); lock(); }
  register_running(_index);
  ret_type _result=_m(...);
  register_finished(_index);
  unlock();
  return _result;
};
```

(2)m无激励条件。

(a)如果m是对父类m的重定义，则不对m重新定义，由父类的m处理条件同步。

(b)如果m是一个新定义的成员函数，则按(1)中的(a)和(b)对m进行重新定义，但在重新定义中没有 while (!B) { unlock(); lock(); }语句。

如果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激励条件的重新定义)，则按上述(1)中的(a)和(b)对m进行重定义，但不产生_m，重新定义的m中将调用父类的_m。

若类或子类中的其它成员函数调用了m，则调用处改成_m，即对象调用自己的成员函数(非私有线程)不受对象条件同步控制的限制。

为了能在子类中使用m的激励条件(通过**<父类名>::activation**)，在对m进行转换时，根据类中是否定义了m的激励条件，另外定义一个成员函数__m：

(1)m有激励条件B。

```
BOOL __m(...)
```

```
{ return B;
}
```

(2)m没有激励条件。

(a)m是对父类m的重定义，则不定义__m;

(b)m是一个新定义的成员函数，则__m定义成：

```
BOOL __m(...)
{ return TRUE;
}
```

当在子类成员函数m的激励条件中出现：**<父类名>::activation**，则把它改写成：<父类名>::__m()。

5.3.2 私有线程的转换

如果t是类C中的一个私有线程，首先把t改名为_t，然后，

(1)把t重新定义为：

```
void t()
{ lock();
  _t();
  unlock();
}
```

(2)定义类C的一个新的静态函数__t：

```
static DWORD __t(LPDWORD p)
{   (C*)p->t();
}
```

(3)把类 C 中对 t 的调用改写成：

```
new_thread(__t);
```

在上述的实现中，C类对象在其某成员函数中通过调用Concorrency类的函数new_thread(__t)产生一个单独的执行线程，并把调用者对象的引用传给该线程。所创建的线程将(异步)执行函数__t()所指定的操作，这里，函数__t()只有一个操作：调用其参数所指定的对象的成员函数t()。成员函数t()没有显式的激励条件，其隐式激励条件为对象处于开放状态。

5.3.3 初始线程的自动执行

为了能在创建对象结束时自动执行对象的初始私有线程init_thread()，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new_thread(__init_thread)，而不是在其父类A的构造函数中调用它(因为在执行父类A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新参数_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。下面的程序片段说明了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) new_thread(__init_thread);
   }
}

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

```
        if (_tag == _CREATE) new_thread(__init_thread);
      };
    }
```

为了在撤消对象时，终止对象所有私有线程的执行，需在并发对象类的析构函数中加入对stop_threads()的调用。

5.3.4 C++类的转换

论文提供的转换程序除了用于实现NDC++的并发对象类以外，还可以用来为已有的C++类加上简单的互斥控制，使得这些类能够运行于并发环境，从而提供并发软件复用顺序软件的一种途径。

由于顺序对象类中没有条件同步描述，转换机制只保证这些类的对象互斥(顺序)地接收消息进行处理。转换中首先给每个C++类加上一个基类Concurrency，然后把类中的成员函数按没有激励条件的情形进行转换(详见5.3.1)。

5.4 Concurrency 类的实现

Concurrency类的实现与具体所使用的操作系统有关，这里，论文给出了一个在Windows 95和NT环境下实现的Concurrency类。

5.4.1 lock()与 unlock()的实现

为了实现lock()与unlock()，论文利用了Windows 95和NT的SDK(下面简称SDK)所提供的功能。首先在Concurrency类中定义一个互斥量mutex：

```
HANDLE mutex;
```

该互斥量在Concurrency类的构造函数中通过调用SDK的CreateMutex进行创建，在Concurrency类的析构函数中调用SDK的CloseHandle把它撤消。在lock()中通过调用了SDK的WaitForSingleObject函数来测试mutex，该函数有个特点：当调用线程获得mutex后，再次调用它不使自己等待。在unlock()中调用ReleaseMutex释放mutex。由于多次调用WaitForSingleObject后，必须调用相同多次ReleaseMutex才能释放mutex，因此，在unlock()中采用了循环调用ReleaseMutex，直到ReleaseMutex返回0。下面给出了Concurrency类中有关互斥量的操作函数的实现：

```
Concurrency()
{  mutex = CreateMutex(NULL,FALSE,NULL);
   .......
}

~Concurrency()
{  CloseHandle(mutex);
   .......
}

void lock();
{ WaitForSingleObject(mutex,INFINITE);
};

void unlock();
{ while (ReleaseMutex(mutex));
};
```

5.4.2 new_thread()和 stop_threads()的实现

为了实现对象私有线程的管理，在Concurrency类中定义了一个线程队列threads_queue。当创建对象的一个私有线程时，该私有线程的handle被放入线程队列；线程执行结束，其handle从线程队列中删除。stop_threads()强行终止线程队列中的所有线程的执行。threads_queue以互斥方式实现。下面给出了new_thread()和stop_threads()实现：

```
void new_thread(FUNC p)
{  DWORD id;
   HANDLE hThread;
   hThread = CreateThread(NULL,0,p,this,CREATE_SUSPEND,&id);
   把新建线程的 hThread(handle)放入对象的线程队列；
   ResumeThread(hThread);
}
```

```
void stop_threads()
{  while (线程队列中有线程)
   { 取第一个线程的 handle: hThread
     TerminateThread(hThread,0);
     把 hThread 从线程队列中删除；
   }
}
```

在lock的实现中通过调用SDK的CreateThread创建一个线程，但该线程并不立即执行，只有在调用ResumeThread之后才开始执行。

5.4.3 与消息有关的函数的实现

为了实现与消息等待与执行有关的一些历史函数，在Concurrency类中定义了两个队列：等待消息队列(waiting_messages_queue)和执行消息队列(running_messages_queue)，它们分别以互斥方式实现。

register_waiting()实现为把消息放入等待消息对列；register_running()把一个消息从等待消息队列中删除，然后放入执行消息队列；register_finished()把消息从执行消息队列中删除。

is_waiting()和waiting_before()的实现基于等待消息队列中的信息；is_running()是根据执行消息队列的信息来实现的。

Concurrency类中与消息有关的函数提供了实现并发对象消息接收调度的一种缺省模型，如果在设计某并发对象类时，Concurrency类中的消息调度模型不能满足需要，则设计者可在所定义的并发对象类中自己设计所需要的消息调度模型。

5.5 NDC++的支撑环境

为了支持用NDC++进行并发面向对象程序设计，论文设计了NDC++的一个支持环境，该环境主要具有下述功能：

(1)语法制导的编辑。

(2)NDC++程序自动转换成C++程序。

(3)图形化的类继承关系察看。

(4)统计信息显示，等等。

5.6 小结

本章根据第三章提出的并发面向对象模型给出了一种对C++进行并发扩充的方案NDC++，该方案采用了一个预定义的类Concurrency和若干新的语言成分。为了使得用扩充后的C++编写的程序能够被C++编译器识别，本章设计了一个转换策略，用于把NDC++程序转换成C++程序。用NDC++设计的并发对象类在使用上与其它C++类的使用完全相同，即先定义并发对象类的实例(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的初始私有线程，它由并发对象类的实现自动完成。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。

第六章 结束语

6.1 论文的主要贡献和特色

现有的并发面向对象模型大多数只考虑系统的表达能力，而忽略了并发与面向对象模型的集成，所引进的并发机制往往与面向对象模型的一些已有特性相冲突，本论文的主要贡献及特色在于：

1. 并发与面向对象模型的集成。

论文从面向对象模型的特点出发，以一种集成的方式把并发引进到面向对象模型中来。这主要体现在：

(1)不区分主动对象与被动对象，模型中只有一种对象：并发对象，它们是自治的并发单位，都具有并发控制能力，可用于各种环境(顺序或并发)。

(2)对象的并发控制由对象自己来完成，不需要其它对象的配合，从而体现了对象的自治性(局部性)和用户透明性(封装性)。

(3)支持并发对象类的继承。

2. 把对象的条件同步控制与内部并发控制分开。

论文把对象的并发控制分成两类：条件同步控制和内部并发控制。条件同步控制用于控制对象消息的接收，实现对象间的并发控制；内部并发控制用于控制对象内部各执行线程对对象局部变量的互斥使用。论文采用的条件同步控制机制支持消息调度，并且，调度策略可由对象类的设计者自行设计，这体现了系统的开放性。

3. 对继承机制的支持。

继承异常是在面向对象模型中引进并发必须要考虑的一个问题，关于继承异常的研究普遍认为：解决继承异常的办法是把对象的并发控制描述与对象的功能性描述分开实现，但要做到这一点往往很困难，因为，如果把这两者完全分开，

则系统的表达能力将受到影响。本论文采取了一种折衷的解决方案，把对继承影响较大的条件同步控制从方法实现中分离出来，这样减轻了继承异常的程度，同时又兼顾了系统的表达能力。另外，论文采用了显式的激励条件继承，即，在定义子类方法的激励条件时可以显式地指定把父类的激励条件作为其布尔表达式中的一部分，从而提高继承机制的表达能力。

4. 实现对象对消息的同步接收、异步处理。

论文通过在对象中引入私有线程实现消息的异步处理，它与通常的异步消息传递的不同之处在于：消息是同步接收、异步处理，这样使得所采用的远程过程调用机制既有同步消息传递机制的同步特性，又有异步消息传递的高并发度，并且，异步执行的程度可由对象类的设计者控制。

5. 给出了模型的一个具体实现NDC++。

6.2 进一步工作

在论文所提出的并发面向对象模型中还有一些值得进一步研究的问题：

(1)解决继承异常的更好方案。
(2)是否需要引进异步消息传递机制。
(3)使对象内部并发控制系统化。

致　谢

首先，感谢我的导师郑国梁教授对本论文工作的精心指导，郑老师不仅在学业上给予我极大的帮助，而且还对我的工作和生活给予了多方面的关心，使得我能安心进行我的博士论文工作。

其次，感谢李宣东博士对本论文工作的关心，特别是在论文的选题和资料的收集方面给予的很大帮助。

第三，感谢袁晓东同学对本论文工作提供的一些建设性意见，并仔细阅读了本论文中的一些算法；感谢赵建华同学在本论文的实现方面提供了一些很好的建议；感谢李勇、严萍同学对我其它工作的支持，使得我有足够的精力进行本论文的工作；感谢教研室的其它老师和同学对我论文工作提供的各种帮助。

最后，感谢我的妻子薛敏女士对我博士研究工作的支持和鼓励。

作者在博士生期间发表(包括已录用)的论文

[1] **Chen Jiajun** and Zheng Guoliang, **NDC++: An Approach to Concurrent Extension of C++**, ACM SIGPLAN Notices, V.32(3), March, 1997.

[2] **陈家骏**、郑国梁，**结构化软件开发过程的平稳过渡**，计算机研究与发展，第33卷，第3期，1996年3月。

[3] **陈家骏**、赵建华、郑国梁，**C++的一种并发扩充方案**，软件学报，已录用

[4] **陈家骏**、袁晓东、郑国梁，**并发对象的一种描述机制**，计算机应用与软件，已录用.

[5] **Chen Jiajun**, Yuan Xiaodong and Zheng Guoliang, **A Multi-Threaded Object-Oriented Programming Model**, ACM Software Engineering Notes, Accepted.

[6] **Chen Jiajun**, Zheng Tao and Zheng Guoliang, **Persistent Objects in the Development of User Interface**, TOOLS Asia'97 & OOT China'97.

[7] **陈家骏**、郑国梁，**C++中行为继承的实现**，信息－电子与自动化仪表，1997年，第4期.

[8] **陈家骏**、王启祥，**日语自动分词的一种实现算法**，微型计算机, Vol 16(6), 1996, 11.

[9] 袁晓东、**陈家骏**、郑国梁，**面向对象方法中的类型概念**，计算机研究与发展，第34卷，第10期，1997年10月。

[10]袁晓东、**陈家骏**、郑国梁，**基于角色分类的子类型关系**，计算机研究与发展，第34卷，第11期，1997年11月。

[11]Yuan Xiaodong, **Chen Jiajun** and Zheng Guoliang, **Two Dimensional Software Development Model Combining Object-Oriented Method with Formal Method**, ACM Software Engineering Notes, Accepted.

[12]袁晓东、**陈家骏**、郑国梁，**COOZ 中的时段演算**，软件学报，1997 年 6 月，增刊。

[13]袁晓东、**陈家骏**、郑国梁，**OOZE 求精技术自动化的探讨**，计算机研究与发展，已录用。

参考文献

[1]B. Achauer, Distribution in Trellis/DOWL, TOOLS USA ’91.

[2]B. Achauer, The DOWL Distributed Object-Oriented Languages, CACM, Vol. 36, No. 9, Sept.,1993.

[3]G. Agha, Concurrent Object-Oriented Programming, CACM, 33(9), Sept., 1990.

[4]G. Agha, et al, Abstraction and Modularity Mechanisms for Concurrent Computing, In G. Agha, etc., editors, Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[5]G. Agha, Foundational Issues in Concurrent Computing, Proc. of ACM SIGPLAN Workshop on Object-Based Concurrent Programming, San Diego, Sept. 26-27, 1988.

[6]G. R. Andrews and F. B. Schneider, Concepts and Notations for Concurrent Programming, Computing Surveys, Vol. 15, No. 1, March 1983

[7]P. America and F.V.D. Linden, A Parallel Object-Oriented Language with Inheritance and Subtyping, ECOOP/OOPSLA '90, 1990.

[8]P. America, Designing an Object-Oriented Programming Language with Behavior Subtyping, LNCS, 489, Springer Verlag, 1991.

[9]P. America, Inheritance and Subtyping in a parallel object-oriented language, Proc. of ECOOP’87, LNCS, 276, Springer Verlag, 1987

[10]P. America, POOL-T: A Parallel Object-Oriented Language, In A. Yonezawa and M. Tokoro, editors, Object-Oriented Concurrent Programming, MIT Press, 1987.

[11]C. Atkinson, Object-Oriented Reuse, Concurrency and Distribution, Addison-Wesley Publishing Company, 1991.

[12]G. Booch, Object-Oriented Analysis and Design, The Benjamin/Cummings Publishing Company, Inc., 1994.

[13]G. Booch, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc,1983.

[14]J. V. D. Bos and C. Laffra, PROCOL: A Parallel Object Language with Protocols, Proc. of OOPSLA ’89.

[15]J. P. Briot and A. Yonezawa, Inheritance and Synchronization in Concurrent OOP, Proc. of ECOOP’87, LNCS, 276, Springer Verlag, 1987

[16]D. E. Brumbaugh, Object-Oriented Development(Building CASE Tools with C++), John Wiley & Sons, Inc, 1994.

[17]R. H. Campbell, et al, Designing and Implementing Choices: An Object-Oriented System in C++, CACM, Vol 36, No 9, Sept.,1993.

[18]R. H. Campbell and A. N. Habermann, The Specification of Process Synchronization by Path Expressions, LNCS 16, Springer Verlag, 1974.

[19]D. Caromel, Toward a Method of Object-Oriented Concurrent Programming, CACM 36(9), Sept.,1993.

[20]D. Caromel and M. Rebuffel, Object-Based Concurrency: Ten Language Features to Achieve Reuse, TOOLS USA '93.

[21]J. J. Chen and G. L. Zheng, NDC++: An Approach to Concurrent Extension of C++, ACM SIGPLAN Notices, Vol. 32(3), March, 1997.

[22]J. J. Chen, X. D. Yuan and G. L. Zheng, A Multi-Threaded Object-Oriented Programming Model, ACM Software Engineering Notes, Accepted.

[23]P. Coad and E. Yourdon, Object-Oriented Design, Yourdon Press, Prentice Hall, 1991.

[24]M. E. Conway, Design of a Separable Transition-Diagram Compiler, CACM, Vol. 6, No 7, July, 1963.

[25]M. E. Conway, A Multiprocessor System Design, In Proc. AFIPS Fall It. Computer Conf.(Las Vegas, Nev., Nov., 1963), Vol. 24. Spartan Books, Baltimore, Maryland.

[26]W. R. Cook, Object-Oriented Programming versus Abstract Data Types, LNCS, 489, Springer Verlag, 1991.

[27]T. B. David, Object-Oriented Design for C++, PTR Prentice Hall, Inc., 1993.

[28]J. B. Dennis and E. C. Van Horn, Programming Semantics for Multiprogrammed Computations, CACM Vol 9, No 3, March, 1966.

[29]E. W. Dijkstra, Cooperating Sequential Processes, In F. Genuys(Ed.), Programming Languages, Academic Press, New York, 1968.

[30]S. Ferenczi, Concurrent Objects with Inherited Synchronizatin,

[31]S. Frolund, Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages, Proceedings of ECOOP'92, 1992.

[32]S. Frolund and G. Agha, A Language Framework for Multi-Object Coordination, ECOOP '93, LNCS 707

[33]N. H. Gehani and W. D. Roome, Concurrent C++: Concurrent Programming with Class(es), Software Practice and Experience, Vol. 18(12), Dec. 1988.

[34]N. H. Gehani and W. D. Roome, Implementing Concurrent C, Software Practice and

Experience, Vol. 22(3), March, 1992.

[35]W. Gerteis and W. Wirz, Synchronizing Objects by Conditional Path Expressions, TOOLS Pacific ’91.

[36]A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[37]R. Guerraoui, Atomic Object Composition, Proc. of ECOOP ’94, LNCS, 821, Springer-Verlag, 1994.

[38]L. Gunaseelan and R. J. LeBlanc, Distributed Eiffel: A Language for Programming Multi-Granular Distributed Objects, Proc. of the 4th International Conference on Computer Languages, IEEE(1992).

[39]P. B. Hansen, Distributed Processes: A Concurrent Programming Concept, CACM, Vol. 21, No. 11, November, 1978.

[40]M. P. Herlihy and J. M. Wing, Linerizability: A Correctness Condition for Concurrent Objects, ACM Trans. on Programming Languages and System, Vol.12, No.3, July 1990.

[41]C. Hewitt, Viewing Control Structures as Patterns of Passing Messages, Journal of Artificial Intelligence, 8(3), 1977.

[42]C. A. R. Hoare, Towards a Theory of Parallel Programming, In C. A. R. Hoare and R. H. Perrott(Eds.), Operating Systems Techniques, Academic Press, New York, 1972.

[43]C. A. R. Hoare, Communicating Sequential Processes, CACM, Vol. 21, No. 8, Aug. 1978.

[44]Y. Ishikawa, Communication Mechanism on Autonomous Objects, Proc. of OOPLSA ’92, ACM SIGPLAN Notices, Vol. 27, No 10, Oct, 1992.

[45]G. Jalloul and J. Potter, Models for Concurrent Eiffel, TOOLS Pacific '91.

[46]S. Janson, J. Montelius and S. Haridi, Ports for Objects in Concurrent Logic Programs, In G. Agha, etc., editors, Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[47]M. Jazayeri, et al, CSP/80: A Language for Communicating Processes, Proc. Fall IEEE COMPCON80(Sept. 1980), IEEE, New York 1980.

[48]J. M. Jezequel, Object-Oriented Software Engineering with Eiffel, Addison-Wesley Publishing Company, Inc., 1996

[49]D. G. Kafura and K. H. Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, ECOOP '89, 1989.

[50]M. Karaorman and J. Bruno, Introducing Concurrency to a Sequential Language, CACM 36(9), Sept.,1993.

[51]R. Lea, Supporting Object-Oriented Languages in a Distributed Environment: The COOL Approach, TOOLS USA ’91.

[52]R. Lea, C. Jacquemot and E. Pillevesse, COOL: System Support for Distributed Programming, CACM, Vol 36, No 9, Sept.,1993.

[53]X. D. Li and G. L. Zheng, Objectpattern: A New Encapsulation Mechanism Enhancing Reusability and Maintainability in Object-Oriented Languages, Proc.of the 20th International Conference on Technology of Object-Oriented Languages and Systems, USA, July, 1996.

[54]J. Lim and R. E. Johnson, The Heart of Object-Oriented Concurrent Programming, Proc. of ACM SIGPLAN Workshop on Object-Based Concurrent Programming, San Diego, Sept. 26-27, 1988.

[55]B. Liskov, Data Abstraction and Hierarchy, OOPSLA ‘87 Addendum to the Proceedings, October, 1987.

[56]K. P. L ö hr, Concurrency Annotations for Reusable Software, CACM 36(9), Sept.,1993.

[57]C. V. Lopes and K. J. Lieberherr, Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications, Proc. of ECOOP ’94, LNCS, 821, Springer-Verlag, 1994.

[58]S. Matsuoka, K. Taura and A. Yonezawa, Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, OOPSLA '93.

[59]S. Matsuoka and A. Yonezawa, Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, In G. Agha, etc., editors, Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[60]S. Matsuoka, Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming, thesis draft, Department of Information Science, The University of Tokyo, April 8, 1993.

[61]C. McHale, B. Walsh, S. Baker and A. Donnelly, Scheduling Predicates, Object-Based Concurrent Computing, LNCS 612, Springer-Verlag, 1992.

[62]J. Meseguer, Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming, Proceedings of ECOOP '93.

[63]B. Meyer, Eiffel: The Language, Prentice Hall, 1991.

[64]B. Meyer, Systematic Concurrent Object-Oriented Programming, CACM 36(9), Sept.,1993.

[65]B. Meyer, Object-Oriented Software Construction, 2nd Ed.

[66]B. Meyer, The Many Faces of Inheritance: A Taxonomy of Taxonomy, Computer. May, 1996, IEEE.

[67]M. Muhlhauser, W. Gerteis and L. Heuser, DOCASE: A Methodic Approach th Distributed Programming, CACM, Vol 36, No 9, Sept.,1993.

[68]O. Nierstrasz, Active Objects in Hybrid, OOPSLA' 87 Proceedings, Oct. 1987

[69]O. Nierstrasz, Composing Active Objects, In G. Agha, etc., editors, Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[70]O. Nierstrasz and M. Papathomas, Viewing Objects as Patterns of Communicating Agents, ECOOP/OOPSLA '90

[71]C. Neusius, Synchronizing Actions, Proc. of ECOOP'91, Geneva, Switzerland, LNCS, 512, Springer Verlag, July, 1991

[72]M. Papathomas, Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming, Ph.D. dissertation, Universite de Geneve, 1992.

[73]G. D. Parrington and S. K. Shrivastava, Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems, Proc. of ECOOP '88.

[74]J. M. Richter, Advanced Windows NT: The Developer's Guide to the Win32 Application Programming Interface, Microsoft Press, 1994.

[75]T. J. Roper and C. J. Barter, A Communicating Sequential Process Language and Implementation, Software Practice&Experience, 11, 1998.

[76]J. Rumbaugh, etc., Object-Oriented Modeling and Design, Prentice Hall, 1991.

[77]M. Sakkinen, On the Darker Side of C++, Proc. of ECOOP '88, LNCS 322, Springer-Verlag, 1988.

[78]M. Sakkinen, Disciplined Inheritance, Proc. of ECOOP '89.

[79]H. Saleh and P. Gautron, A Concurrent Control Mechanism for C++ Objects, Object-Based Concurrent Computing, LNCS 612, Springer-Verlag, 1992.

[80]C. R. Snow, Concurrent Programming, Cambridge University Press, 1992.

[81]D. Steel, Distributed Object Oriented Programming: Mechanism & Experience, TOOL USA '91.

[82]B. Stroustrup, The C++ Programming Language, Addison-Wesley, 1986.

[83]A. Synder, Encapsulation and Inheritance in Object-Oriented Programming

Languages, OOPSLA'86 Proceedings.

[84]L. Thomas, Extensibility and Reuse of Object-Oriented Synchronization Components, LNCS v.605, June, 1992.

[85]C. Tomlinson and V. Singh, Inheritance and Synchrozization with Enabled-Sets, Proc. of OOPSLA'89, Vol. 24, pp. 103-112. SIGPLAN Notices, ACM Press, Oct. 1989.

[86]V. T. Vasconcelos, Typed Concurrent Objects, Proc. of ECOOP '94, LNCS, 821, Springer-Verlag, 1994.

[87]P. Wegner, Design Issues for Object-Based Concurrency, ECOOP '91 Workshop, Object-Based Concurrent Computing, LNCS 612, Springer-Verlag, 1992.

[88]P. Wegner, Granularity of Modules in Object-Based Concurrent Systems, Proc. of ACM SIGPLAN Workshop on Object-Based Concurrent Programming, San Diego, Sept. 26-27, 1988.

[89]P. Wegner, Dimensions of Object-Based Language Design, Proc. of OOPSLA'87, SIGPLAN Notices, Vol. 22, pp. 168-182, ACM, Orlando, Florida, December, 1987

[90]P. Wegner and S. B. Zdonik, Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like, Proceedings of ECOOP '88, LNCS 322, Springer-Verlag, 1988.

[91]Y. Yokote and M. Tokoro, Concurrent Programming in ConcurrentSmalltalk, In A. Yonezawa and M. Tokoro, editors, Object-Oriented Concurrent Programming, MIT Press, 1987.

[92]A. Yonezawa, etc, Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, In A. Yonezawa and M. Tokoro, editors, Object-Oriented Concurrent Programming, MIT Press, 1987.

[93]徐家福、王志坚、翟成祥，对象式程序设计语言，南京大学出版社，1992.

[94]陆汝钤，计算机语言的形式语义，科学出版社，1992。

[95]陈家骏、赵建华、郑国梁，C++的一种并发扩充方案，软件学报，已录用

[96]陈家骏、袁晓东、郑国梁，并发对象的一种描述机制，计算机应用与软件，已录用。

.

附录：NDC++扩充语法

本附录只给出NDC++中对C++扩充部分的语法描述。

<类成员表>::=<C++类成员表>[<条件同步定义>][<私有线程定义>]
<条件同步定义>::=**concurrency_control** <条件同步项表>
<条件同步项表>::=<条件同步项> {<条件同步项>}
<条件同步项>::=<函数定义首部>: [<激励条件>];
<激励条件>::=<布尔表达式>
<布尔表达式>::= ..... <简单表达式> .....
<简单表达式>::=<常量>|<成员变量>|<成员函数形参>|<成员函数调用>|
<类名>::**activation**
<私有线程定义>::=**threads** <私有线程表>
<私有线程表>::=<私有线程> {<私有线程>}
<私有线程>::=void <函数名>() <函数体>;