**第1章 ACE自适配通信环境**

　　ACE自适配通信环境 (Adaptive Communication Environment)是面向对象的构架和工具包，它为通信软件实现了核心的并发和分布式模式。ACE包含的多种组件可以帮助通信软件的开发获得更好的灵活性、效率、可靠性和可移植性。ACE中的组件可用于以下几种目的：

* 并发和同步
* 进程间通信

(IPC)

* 内存管理
* 定时器
* 信号
* 文件系统管理
* 线程管理
* 事件多路分离和处理器分派
* 连接建立和服务初始化
* 软件的静态和动态配置、重配置
* 分层协议构建和流式构架
* 分布式通信服务：名字、日志、时间同步、事件路由和网络锁定，等等。

**1.1 ACE体系结构**

　　如图1-1所示，ACE具有分层的体系结构。在ACE构架中有三个基本层次：

* 操作系统（

OS）适配层

* C++

包装层

* 构架和模式层

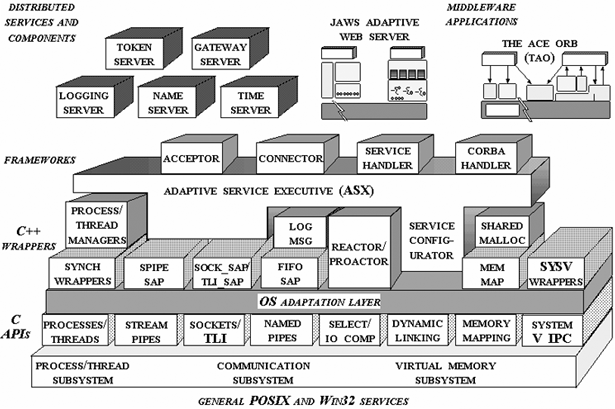


图1-1 ACE的体系结构

**1.1.1 OS适配层**

　　OS适配层是位于本地OS API和ACE之间的“瘦”代码层，它使ACE的较高层与平台依赖性屏蔽开来，从而使得通过ACE编写的代码保持了相对的平台无关性。只需要极少的努力，开发者就可以将ACE应用移植到任何平台上。

　　OS适配层也是ACE构架之所以可用于如此多的平台的原因所在。目前ACE适用的OS平台包括：实时OS（VxWorks、Chorus、LynxOS和pSoS）、大多数版本的UNIX（SunOS 4.x和5.x; SGI IRIX 5.x和6.x; HP-UX 9.x, 10.x和11.x; DEC UNIX 3.x和4.x; AIX 3.x和4.x; DG/UX; Linux; SCO; UnixWare; NetBSD和FreeBSD）、Win32（使用MSVC++和Borland C++的WinNT 3.5.x、4.x、Win95和WinCE）以及MVS OpenEdition。

**1.1.2 C++包装层**

　　C++包装层包括一些C++包装类，它们可用于构建高度可移植的和类型安全的C++应用。这是ACE工具包最大的一部分，大约包含了总源码的50%。C++包装类可用于：

* **并发和同步：**

ACE提供若干并发和同步包装类，对本地OS多线程和多进程API进行了抽象。这些包装类封装用于线程和进程的原语，比如信号量、锁、栅栏（Barrier）和条件变量。另外还有更高级的原语可用，比如守卫（Guard）。所有这些原语共享类似的接口，因而很容易使用和相互替换。

* **IPC**

**：**ACE提供若干C++包装类，封装不同OS中不同的进程间通信（IPC）接口。例如，ACE的包装类封装了以下IPC机制：BSD socket、TLI、UNIX FIFO、流管道、Win32命名管道，等等。ACE还为消息队列提供包装类，包括特定的实时OS的消息队列。

* **内存管理组件：**

ACE包含的一些类可用于内存动态分配和释放；其中包括允许预分配所有动态内存的类。这些预分配的内存随即通过ACE提供的管理类的帮助进行本地管理。在大多数实时和嵌入式系统中，这样的细粒度管理极为必要。另外还有一些类用于灵活地管理进程间共享内存。

* **定时器类：**有多种不同的类可用于处理定时器的调度和取消。

ACE中不同种类的定时器使用不同的底层机制（堆、定时器轮（timer wheel）或简单列表）来提供不同的性能特性。但是，不管底层使用何种机制，这些类的接口都是一致的，从而使得开发者很容易使用任何一种定时器类。除了这些定时器类，还有封装高分辨率定时器（在部分平台上可用，比如VxWorks, Win32/Pentium, AIX和Solaris）和Profile Timer的包装类。

* **容器类：**

ACE还拥有若干可移植的STL风格的容器类，比如Map、Hash\_Map、Set、List，等等

* **信号处理：**

ACE提供对特定OS的信号处理接口进行封装的包装类。这些类使得开发者能够很容易地安装和移除信号处理器，并且可以为一个信号安装若干处理器。另外还有信号守卫类，可用于在看守的作用域之内禁止所有信号。

* **文件系统组件：**

ACE含有包装文件系统API的类。这些类包括文件I/O、异步文件I/O、文件加锁、文件流、文件连接包装，等等。

* **线程管理：**

ACE提供包装类来创建和管理线程。这些包装还封装了针对特定OS的线程API，可被用于提供像线程专有存储这样的功能。

**1.1.3 ACE构架组件**

　　ACE构架组件是ACE中最高级的“积木”，它们的基础是若干针对特定通信软件领域的设计模式。设计者可以使用这些构架组件来帮助自己在高得多的层面上思考和构建系统。这些组件实际上为将要构建的系统提供了“袖珍体系结构”，因此这些组件不仅在开发的实现阶段、同时在设计阶段都是有用的。ACE的这一层含有以下一些大型组件：

* **事件处理：**

大多数通信软件都含有大量处理各种类型事件（比如，基于I/O、基于定时器、基于信号和基于同步的事件）的代码。软件必须高效地多路分离、分派和处理这些事件。遗憾的是，大多数时间开发者们都在反复地编写这些代码，“重新发明轮子”。这是因为，事件多路分离、分派和处理代码全都紧密地耦合在一起，无法彼此独立地使用。ACE提供了被称为**Reactor**（反应堆）的构架组件来解决这一问题。反应堆提供用于高效地进行事件多路分离和分派的代码，并极大地降低了它们与处理代码之间的耦合，从而改善了可复用性和灵活性。

* **连接或服务初始化组件：**

ACE提供**Connector**（连接器）和**Acceptor**（接 受器）组件，用于降低连接初始化与连接建立后应用执行的实际服务之间的耦合。在接受大量连接请求的应用服务器中，该组件十分有用。连接首先以应用特有的方 式初始化，然后每一连接被相应的处理例程分别处理。这样的去耦合使得开发者能够分别去考虑连接的处理和初始化。因此，如果在随后的阶段开发者发现连接请求 数多于或是少于估算，它可以选择使用不同的初始化策略集（ACE提供了若干可供开发者挑选和选择的策略），以获得所要求的性能水平。

* **流组件：**

ACE **Stream**组件用于简化那些本质上是分层的（layered）或层次的（hierarchic）软件的开发。用户级协议栈的开发是一个好例子；这样的栈由若干互连的层次组成。这些层次或多或少可以相互独立地进行开发。当“数据”通过时，每一层都处理并改变它，并将它传递给下一层，以作进一步的处理。因为各层是相互独立的，它们很容易被复用或替换。

* **服务配置组件：**通信软件开发者面临的另一个问题是，很多时候，软件服务必须在安装时配置，或必须在运行时重配置。应用中特定服务的实现可能需要进行*改动*，因而应用可能必须用新改动的服务重新配置。

ACE **Service Configurator**（服务配置器）为应用的服务提供动态的启动、挂起和配置。

　　尽管计算机网络领域发展迅速，编写通信软件已经变得更为困难。大量消耗在开发通信软件上的努力不过是“重新发明轮子”的变种，已知的可以在应用间通用的组件被重写，而不是被复用。通过收集通用的组件和体系结构（它们在网络和系统编程领域一再被复用），ACE为这一问题提供了解决方案。应用开发者可以采用ACE，挑选和选择在他的应用中所需的组件，并开始在ACE工具箱的陪伴下构建应用。除了在C++包装层中收集简单的“积木”，ACE还包括了大的体系结构“积木”，它们采用了已被证明在软件开发领域中行之有效的“模式”和“软件体系结构”。

**第2章　IPC SAP：进程间通信服务访问点包装**

　　socket、TLI、STREAM管道和FIFO为访问局部和全局IPC机制提供广泛的接口。但是，有许多问题与这些不统一的接口有关联。比如类型安全的缺乏和多维度的复杂性会导致成问题的和易错的编程。

　　ACE的IPC SAP类属提供了统一的层次类属，对那些麻烦而易错的接口进行封装。在保持高性能的同时，IPC SAP被设计用于改善通信软件的*正确性*、*易学性*、*可移植性*和*可复用性*。

**2.1 IPC SAP类属**

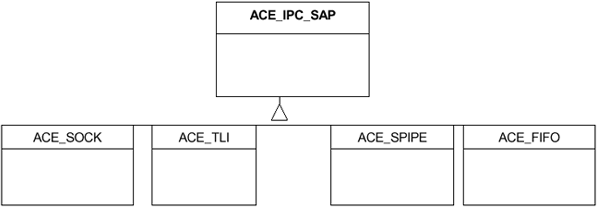


图2-1 IPC SAP类属

　　根据底层使用的不同IPC接口，IPC SAP类被划分为四种主要的类属，图2-1描述了这一划分。ACE\_IPC\_SAP类提供的一些函数是所有IPC接口公有的。有四个不同的类由此类派生而出，每个类各自代表ACE包含的一种IPC SAP包装类属。这些类封装适用于特定IPC接口的功能。例如，ACE\_SOCK类包含的功能适用于BSD socket编程接口，而ACE\_TLI包装TLI编程接口。

　　在这四个类的每一个类下面都有一整层次的包装类，它们完整地包装底层接口，并提供高度可复用、模块化、安全和易用的包装类。

**2.2 socket类属（ACE\_SOCK）**

　　该类属中的类都位于ACE\_SOCK之下；它提供使用BSD socket编程接口的Internet域和UNIX域协议族的接口。这个类属中的类被进一步划分为：

* Dgram

类和Stream类：Dgram类基于UDP数据报协议，提供不可靠的无连接消息传递功能。另一方面，*Stream*类基于TCP协议，提供面向连接的消息传递。

* Acceptor

、Connector类和Stream类：Acceptor和Connector类分别用于被动和主动地建立连接。Acceptor类封装BSD accept()调用，而Connector封装BSD connect()调用。Stream类用于在连接建立**之后**提供双向的数据流，并包含有发送和接收方法。

　　表2-1详细描述了该类属中的类以及它们的职责：

|  |  |
| --- | --- |
| **类名** | **职责** |
| ACE\_SOCK\_Acceptor | 用于被动的连接建立，基于BSD accept()和listen()调用。 |
| ACE\_SOCK\_Connector | 用于主动的连接建立，基于BSD connect()调用。 |
| ACE\_SOCK\_Dgram | 用于提供基于UDP（用户数据报协议）的无连接消息传递服务。封装了sendto()和receivefrom()等调用，并提供了简单的send()和recv()接口。 |
| ACE\_SOCK\_IO | 用于提供面向连接的消息传递服务。封装了send()、recv()和write()等调用。该类是ACE\_SOCK\_Stream和ACE\_SOCK\_CODgram类的基类。 |
| ACE\_SOCK\_Stream | 用于提供基于TCP（传输控制协议）的面向连接的消息传递服务。派生自ACE\_SOCK\_IO，并提供了更多的包装方法。 |
| ACE\_SOCK\_CODgram | 用于提供有连接数据报（connected datagram）抽象。派生自ACE\_SOCK\_IO；它包含的open()方法使用bind()来绑定到指定的本地地址，并使用UDP连接到远地地址。 |
| ACE\_SOCK\_Dgram\_Mcast | 用于提供基于数据报的多点传送(multicast)抽象。包括预订多点传送组，以及发送和接收消息的方法 |
| ACE\_SOCK\_Dgram\_Bcast | 用于提供基于数据报的广播(broadcast)抽象。包括在子网中向所有接口广播数据报消息的方法 |

表2-1 ACE\_SOCK中的类及其职责

　　在下面的部分，我们将要演示怎样将IPC\_SAP包装类直接用于处理进程间通信。记住这些只是ACE的冰山一角。在教程的后续章节中将会介绍其他类和组件。

**2.2.1 使用ACE的流**

　　ACE中的流包装提供面向连接的通信。流数据传输包装类包括ACE\_SOCK\_Stream和ACE\_LSOCK\_Stream，它们分别包装TCP/IP和UNIX域socket协议数据传输功能。连接建立类包括针对TCP/IP的ACE\_SOCK\_Connector和ACE\_SOCK\_Acceptor，以及针对UNIX域socket的ACE\_LSOCK\_Connector和ACE\_LSOCK\_Acceptor。

　　Acceptor类用于被动地接受连接（使用BSD accept()调用），而Connector类用于主动地建立连接（使用BSD connect()调用）。

　　下面的例子演示接收器和连接器是怎样用于建立连接的。该连接随后将用于使用流数据传输类来传输数据。

Server类包含的accept\_connections()方法使用接受器（也就是ACE\_SOCK\_Acceptor）来将连接接受“进”ACE\_SOCK\_Stream new\_stream\_。该操作是这样来完成的：调用接受器上的accept()，并将流作为参数传入其中；我们想要接受器将连接接受进这个流。一旦连接已建立进流中，流的包装方法send()和recv()就可以用来在新建立的链路上发送和接收数据。还有一个空的ACE\_INET\_Addr也被传入接受器的accept()方法，并在其中被设定为发起连接的远地机器的地址。

　　在连接建立后，服务器调用handle\_connection()方法，它开始从客户那里读取一个预先知道的单词，然后将流关闭。对于要处理多个客户的服务器来说，这也许并不是很实际的情况。在现实世界的情况中可能发生的是，连接在单独的线程或进程中被处理。在后续章节中将反复演示怎样完成这样的多线程和多进程类型的处理。

　　连接关闭通过调用流上的close()方法来完成，该方法会释放所有的socket资源并终止连接。

　Connect\_to\_server()方法使用类型为ACE\_SOCK\_Connector的连接器（connector\_）来主动地建立连接。连接的设置通过调用连接器connector\_上的connect()方法来完成：传入的参数为我们想要连接的机器的地址remote\_addr\_，以及用于在其中建立连接的空ACE\_SOCK\_Stream client\_stream\_。远地机器在例子的运行时参数中指定。一旦connect()方法成功返回，通过使用ACE\_SOCK\_Stream封装类中的send()和recv()方法族，流就可以用于在新建立的链路上发送和接收数据了。

　　在此例中，一旦连接建立好，send\_to\_server()方法就会被调用，以将一个字符串发送NO\_ITERATIONS次到服务器。如前面所提到的，这是通过使用流包装类的send()方法来完成的。

**2.2.2 使用ACE的数据报**

　　ACE\_SOCK\_Dgram和ACE\_LSOCK\_Dgram是ACE中的数据报包装类。这些包装包含了发送和接收数据报的方法，并包装了非面向连接的UDP协议和UNIX域socket协议。与流包装不同，这些包装封装的是非面向连接的协议。这也就意味着不存在用于“设置”连接的接受器和连接器。相反，在这种情况下，通信通过一系列的发送和接收来完成。每个send()都要指定目的远地地址作为参数。下面的例子演示怎样通过ACE使用数据报。这个例子使用了ACE\_SOCK\_Dgram包装（也就是UDP包装）。还可以使用包装UNIX域数据报的ACE\_LSOCK\_Dgram。两种包装的用法非常类似，唯一的不同是ACE\_LSOCK\_Dgram要用 ACE\_UNIX\_Addr类作为地址，而不是ACE\_INET\_Addr。

**2.2.3 使用ACE的多点传送（Multicast）**

　 　你会发现，在许多场合，同样的消息必须被发送给你的分布式系统中的众多客户或服务器。例如，可能需要将时间调整更新或其他的周期性信息广播给特定的终端 集。多点传送被用于处理这一问题。它允许对特定的终端子集或组、而不是所有终端进行广播。因此，你可以认为多点传送是一种受控的广播机制。大多数现代OS都提供多点传送功能。

　　ACE提供ACE\_SOCK\_Dgram\_Mcast包装，封装了不可靠的多点传送。它允许程序员将数据报消息发送给被称为“多点传送组”的受控组。这样的组由唯一的多点传送地址标识。

　 　对在此地址上接收广播有兴趣的客户和服务器必须进行预订（也被称为“多点传送组预订”）。于是，所有预订到此多点传送组的进程将会接收到所有发送给该组 的数据报消息。仅仅想要给多点传送组发送消息，而不需要收听它们的应用，无需进行预订。实际上，这样的发送者可以使用原有的简单ACE\_SOCK\_Dgram包装给多点传送地址发送消息，整个多点传送组将随之收到发送出的消息。

　　在ACE中，多点传送功能被封装在ACE\_SOCK\_Dgram\_Mcast中，其中包括在多点传送组上的预订、取消预订和接收功能。

**第3章 ACE的内存管理**

　　ACE构架含有一组非常丰富的内存管理类。这些类使得你能够很容易和有效地管理动态内存（从堆中申请的内存）和共享内存（在进程间共享的内存）。你可以使用若干不同的方案来管理内存。你需要决定何种方案最适合你正在开发的应用，然后采用恰当的ACE类来实现此方案。

　　ACE含有两组不同的类用于内存管理。

　　第一组是那些基于ACE\_Allocator的类。这组类使用动态绑定和策略模式来提供灵活性和可扩展性。它们只能用于局部的动态内存分配。

　　第二组类基于ACE\_Malloc模板类。这组类使用C++模板和*外部多态性* （External Polymorphism）来为内存分配机制提供灵活性。在这组类中的类不仅包括了用于局部动态内存管理的类，也包括了管理进程间共享内存的类。这些共享内存类使用底层OS（OS）共享内存接口。

　　为什么使用一组类而不是另外一组呢？这是由在性能和灵活性之间所作的权衡决定的。因为实际的分配器对象可以在运行时改变，ACE\_Allocator类更为灵活。这是通过动态绑定（这在C++里需要使用虚函数）来完成的，因此，这样的灵活性并非不需要代价。虚函数调用带来的间接性使得这一方案成了更为昂贵的选择。

　　另一方面，ACE\_Malloc类有着更好的性能。在编译时，malloc类通过它将要使用的内存分配器进行配置。这样的编译时配置被称为“外部多态性”。基于ACE\_Malloc的分配器不能在运行时进行配置。尽管ACE\_Malloc效率更高，它不像ACE\_Allocator那样灵活。

**3.1 分配器(Allocator)**

　　分配器用于在ACE中提供一种动态内存管理机制。在ACE中有若干使用不同策略的分配器可用。这些不同策略提供相同的功能，但是具有不同的特性。例如，在实时系统中，应用可能必须从OS那里预先分配所有它将要用到的动态内存，然后在内部对分配和释放进行控制。这样，分配和释放例程的性能就是高度可预测的。

　　所有的分配器都支持ACE\_Allocator接口，因此无论是在运行时还是在编译时，它们都可以很容易地相互替换。这也正是灵活性之所在。所以，ACE\_Allocator可以与策略模式协同使用，以提供非常灵活的内存管理。表3-1给出了对ACE中可用的各种分配器的简要描述。这些描述规定了每种分配器所用的内存分配策略。

|  |  |
| --- | --- |
| **分配器** | **描述** |
| ACE\_Allocator | ACE中的分配器类的接口类。这些类使用继承和动态绑定来提供灵活性。 |
| ACE\_Static\_Allocator | 该分配器管理固定大小的内存。每当收到分配内存的请求时，它就移动内部指针、以返回内存chunk（“大块”）。它还假定内存一旦被分配，就再也不会被释放。 |
| ACE\_Cached\_Allocator | 该分配器预先分配内存池，其中含有特定数目和大小的内存chunk。这些chunk在内部空闲表（free list）中进行维护，并在收到内存请求（malloc()）时被返回。当应用调用free()时，chunk被归还到内部空闲表、而不是OS中。 |
| ACE\_New\_Allocator | 为C++ new和delete操作符提供包装的分配器，也就是，它在内部使用new和delete操作符，以满足动态内存请求。 |

表3-1 ACE中的分配器

**第4章 线程管理：ACE的同步和线程管理机制**

　　ACE拥有许多不同的用于创建和管理多线程程序的类。在这一章里，我们将查看ACE中的一些线程管理机制。在一开始，我们将查看那些简单的线程包装类，它们的管理功能很少。但是，随着内容的进展，我们将查看ACE\_Thread\_Manager中的更为强大的管理机制。ACE还拥有一组非常全面的处理线程同步的类。这些类也将在本章讲述。

**4.1 创建和取消线程**

　　在不同的平台上，有着若干不同的用于线程管理的接口。其中包括POSIX pthreads接口、Solaris线程、Win32线程等等。这些接口提供了相同或是相似的功能，但是它们的API的差别却极为悬殊。这就导致了困难、麻烦和易错的编程，因为应用程序员必须熟悉不同平台上的若干接口。而且，这样写下的程序，是不可移植和不灵活的。

　　ACE\_Thread提供了对OS的线程调用的简单包装，这些调用处理线程创建、挂起、取消和删除等问题。它提供给应用程序员一个简单易用的接口，可以在不同的线程API间移植。ACE\_Thread是非常“瘦”的包装，有着很少的开销。其大多数方法都是内联的，因而等价于对底层OS专有线程接口的直接调用。ACE\_Thread中的所有方法都是静态的，而且该类一般不进行实例化。

**4.2 ACE同步原语**

　　ACE有若干可用于同步目的的类。这些类可划分为以下范畴：

* ACE Lock

类属

* ACE Guard

类属

* ACE Condition

类属

* 杂项

ACE Synchronization类

**4.2.1 ACE Lock类属**

　　锁类属包含的类包装简单的锁定机制，比如互斥体、信号量、读／写互斥体和令牌。在这一类属中可用的类在表4-1中显示。每个类名后都有对用法和用途的简要描述：

|  |  |
| --- | --- |
| **名字** | **描述** |
| ACE\_Mutex | 封装互斥机制（根据平台，可以是mutex\_t、pthread\_mutex\_t等等）的包装类，用于提供简单而有效的机制来使对共享资源的访问序列化。它与二元信号量（binary semaphore）的功能相类似。可被用于线程和进程间的互斥。 |
| ACE\_Thread\_Mutex | 可用于替换ACE\_Mutex，专用于线程同步。 |
| ACE\_Process\_Mutex | 可用于替换ACE\_Mutex，专用于进程同步。 |
| ACE\_NULL\_Mutex | 提供了ACE\_Mutex接口的“无为”（do-nothing）实现，可在不需要同步时用作替换。 |
| ACE\_RW\_Mutex | 封装读者／作者锁的包装类。它们是分别为读和写进行获取的锁，在没有作者在写的时候，多个读者可以同时进行读取。 |
| ACE\_RW\_Thread\_Mutex | 可用于替换ACE\_RW\_Mutex，专用于线程同步。 |
| ACE\_RW\_Process\_Mutex | 可用于替换ACE\_RW\_Mutex，专用于进程同步。 |
| ACE\_Semaphore | 这些类实现计数信号量，在有固定数量的线程可以同时访问一个资源时很有用。在OS不提供这种同步机制的情况下，可通过互斥体来进行模拟。 |
| ACE\_Thread\_Semaphore | 应被用于替换ACE\_Semaphore，专用于线程同步。 |
| ACE\_Process\_Semaphore | 应被用于替换ACE\_Semaphore，专用于进程同步。 |
| ACE\_Token | 提供“递归互斥体”（recursive mutex），也就是，当前持有某令牌的线程可以多次重新获取它，而不会阻塞。而且，当令牌被释放时，它确保下一个正阻塞并等待此令牌的线程就是下一个被放行的线程。 |
| ACE\_Null\_Token | 令牌接口的“无为”（do-nothing）实现，在你知道不会出现多个线程时使用。 |
| ACE\_Lock | 定义锁定接口的接口类。一个纯虚类，如果使用的话，必须承受虚函数调用开销。 |
| ACE\_Lock\_Adapter | 基于模板的适配器，允许将前面提到的任意一种锁定机制适配到ACE\_Lock接口。 |

表4-1 ACE锁类属中的类

表4-1中描述的类都支持同样的接口。但是，在任何继承层次中，这些类都是**互不关联**的。在ACE中，锁通常用模板来参数化，因为，在大多数情况下，使用虚函数调用的开销都是不可接受的。使用模板使得程序员可获得相当程度的灵活性。他可以在编译时（但不是在运行时）选择他想要使用的的锁定机制的类型。然而，在某些情形中，程序员仍可能需要使用动态绑定和替换（substitution）；对于这些情况，ACE提供了ACE\_Lock和ACE\_Lock\_Adapter类。

**4.2.1.1 使用互斥体类**

　　互斥体实现了“*互相排斥*”（mutual exclusion）同步的简单形式（所以名为互斥体(mutex)）。互斥体禁止多个线程同时进入受保护的代码“*临界区*”（critical section）。因此，在任意时刻，只有一个线程被允许进入这样的代码保护区。

　　任何线程在进入临界区之前，必须*获取*（acquire）与此区域相关联的互斥体的所有权。如果已有另一线程拥有了临界区的互斥体，其他线程就不能再进入其中。这些线程必须等待，直到当前的属主线程*释放*（release）该互斥体。

　　什么时候需要使用互斥体呢？互斥体用于保护共享的易变代码，也就是，全局或静态数据。这样的数据必须通过互斥体进行保护，以防止它们在多个线程同时访问时损坏。

**4.2.1.3 使用令牌（Token）**

　　如表4-1中所提到的，ACE\_Token类提供所谓的“递归互斥体”，它可以被最初获得它的同一线程进行多次重新获取。ACE\_Token类还确保所有试图获取它的线程按严格的FIFO（先进先出）顺序排序。

　　递归锁允许同 一线程多次获取同一个锁。线程不会因为试图获取它已经拥有的锁而死锁。这些类型的锁能在各种不同的情况下派上用场。例如，如果你用一个锁来维护跟踪流的一 致性，你可能希望这个锁是递归的，因为某个方法可以调用一个跟踪例程，获取锁，被信号中断，然后再尝试获取这个跟踪锁。如果锁是非递归的，线程将会在这里 锁住它自己。你会发现很多其他需要递归锁的有趣应用。重要的是要记住，你获取递归锁多少次，就**必须释放**它多少次。

　　在SunOS 5.x上运行例4-3，释放锁的线程常常也是重新获得它的线程（大约90%的情况是这样）。但是如果你采用ACE\_Token类作为锁定机制来运行这个例子，每个线程都会轮流获得令牌，然后有序地把机会让给下一个线程。

　　尽管ACE\_Token作为所谓的递归锁非常有用，它们实际上是更大的“令牌管理”构架的一部分。该构架允许你维护数据存储中数据的一致性。遗憾的是，这已经超出了此教程的范围

**4.2.2 ACE守卫（Guard）类属**

　　ACE中的守卫用于自动获取和释放锁。守卫类的对象定义一个代码块，在其上获取一个锁。在退出此代码块时，锁被自动释放。

　　ACE中的守卫类是一种模板，它通过所需锁定机制的类型来参数化。底层的锁可以是ACE Lock类属中的任何类，也就是，任何互斥体或锁类。它是这样工作的：对象的构造器获取锁，析构器释放锁。表4-2列出了ACE中可用的守卫：

|  |  |
| --- | --- |
| **名字** | **描述** |
| ACE\_Guard | 自动在底层锁上调用acquire()和release()。任何ACE Lock类属中的锁都可以作为它的模板参数传入。 |
| ACE\_Read\_Guard | 自动在底层锁上调用acquire()和release()。 |
| ACE\_Write\_Guard | 自动在底层锁上调用acquire()和release()。 |

表4-2 ACE中的守卫

守卫在工作者线程中管理临界区。守卫对象从ACE\_Guard模板类创建而来。守卫应使用的锁的类型被传给模板。通过将所需获取的实际锁对象经由守卫对象的构造器传入，程序创建守卫对象。该锁由ACE\_Guard在内部自动获取，所以for循环中的区域就是受保护的临界区。一旦出了作用域，看守对象就会被自动删除，锁也随之被释放。

　　守卫是很有用的，因为它们保证一旦你获取了一个锁，你总是会释放它（当然，除非你的线程因为无法预料的情况而死掉）。在有许多不同的返回路径的复杂方法中，这被证明为是极其有用的。

**4.2.3 ACE条件（Condition）类属**

　　ACE\_Condition类是针对OS条件变量原语的包装类。那么，到底什么是条件变量呢？

　　线程常常需要特定条件被满足才能继续它的操作。例如，设想线程需要在全局消息队列里插入消息。在插入任何消息之前，它必须检查在消息队列里是否有空闲空间。如果消息队列在“满”状态，它就什么也不能做，而必须进行休眠，过一会再重试。就是说，在访问全局资源之前，某个*条件*必须为真。然后，当另外的线程空出消息队列时，应该有方法通知或发信号给原来的线程：在消息队列里有位置了，现在应该再次尝试插入消息。这可以使用条件变量来完成。条件变量不是被用作互斥原语，而是用作特定条件已经满足的指示器。

　　在使用条件变量时，你的程序应该完成以下步骤：

* 获取全局资源（例如，消息队列）的锁（互斥体）。
* 检查条件（例如，消息队列里有空间吗？）。
* 如果条件失败，调用条件变量的

wait()方法。等待在未来条件变为真。

* 当另一线程在全局资源上执行操作时，它发信号（

signal()）给所有其他在此资源上测试条件的线程（例如，另一线程从消息队列中取出一个消息，然后通过条件变量发送信号，以使阻塞在wait()上的线程能够再尝试将它们的消息插入队列）。

* 在醒来之后，重新检查条件现在是否为真。如为真，则在全局资源上执行操作（例如，将消息插入全局消息队列）

　　需要特别注意的是，在阻塞在wait调用中之前，条件变量机制（也就是ACE\_Cond）负责释放全局资源上的互斥体。如果没有进行此操作，将没有其他的线程能够在此资源上工作（该资源是条件改变的原因）。同样，一旦阻塞线程收到信号、重又醒来，它在检查条件之前会在内部重新获取锁。

**4.2.4.1 ACE中的栅栏（Barrier）**

　 　栅栏有一个好名字，因为它恰切地描述了栅栏应做的事情。一组线程可以使用栅栏来进行共同的相互同步。组中的每个线程各自执行，直到到达栅栏，就阻塞在那 里。在所有相关线程到达栅栏后，它们就全部继续它们的执行。就是说，它们一个接一个地阻塞，等待其他的线程到达栅栏；一旦所有线程都到达了它们的执行路径 中的“栅栏点”，它们就一起重新启动。

　　在ACE中，栅栏在ACE\_Barrier类中实现。在栅栏对象被实例化时，它将要等待的线程的数目会作为参数传入。一旦到达执行路径中的“栅栏点”，每个线程都在栅栏对象上发出wait()调用。它们在这里阻塞，直到其他线程到达它们各自的“栅栏点”，然后再一起继续执行。当栅栏从相关线程那里接收了适当数目的wait()调用时，它就同时唤醒所有阻塞的线程。

**4.2.4.2 原子操作（Atomic Op）**

　　ACE\_Atomic\_Op类用于将同步透明地参数化进基本的算术运算中。ACE\_Atomic\_Op是一种模板类，锁定机制和需要参数化的类型被作为参数传入其中。ACE是这样来实现此机制的：重载所有算术操作符，并确保在操作前获取锁，在操作后释放它。运算本身被委托给通过模板传入的的类。

**4.3 使用ACE\_THREAD\_MANAGER进行线程管理**

　　在前面所有的例子中，我们一直使用ACE\_Thread包装类来创建和销毁线程。但是，该包装类的功能比较有限。ACE\_Thread\_Manager提供了ACE\_Thread中的功能的超集。特别地，它增加了管理功能，以使启动、取消、挂起和恢复一组相关线程变得更为容易。它用于创建和销毁成组的线程和任务（ACE\_Task是一种比线程更高级的构造，可在ACE中用于进行多线程编程。我们将在后面再来讨论任务）。它还提供了这样的功能：发送信号给一组线程，或是在一组线程上等待，而不是像我们在前面的例子中所看到的那样，以一种不可移植的方式来调用join()。

**4.4 线程专有存储（Thread Specific Storage）**

　 　如果单线程程序希望创建一个变量，其值能在多个函数调用间持续，它可以静态或全局地分配该数据。如果这是个多线程程序，这个全局或静态数据对于所有线程 来说都是一样的。这可能是，也可能不是我们所期望的。例如，伪随机数生成器可能需要静态的或全局的整型种子变量，它不会因为多个线程同时改变它的值而受到 影响。但是，在另外的情形中，对于各个线程来说，可能需要不同的全局或静态数据。例如，设想一个多线程的GUI应 用，在其中各个窗口都运行在独立的线程中，并各拥有一个输入端口，窗口从中接收事件输入。这样的输入端口必须在窗口的各个函数调用之间保持“持续”，并且 还必须是窗口专有的或私有的。可使用线程专有存储来满足此需求。像输入端口这样的结构可放在线程专有存储中，并可像逻辑上的静态或全局变量一样被访问；而 实际上它对线程来说是私有的。

　　传统上，线程专有存储通过让人迷惑的底层操作系统API来实现。在ACE中，TSS通过使用ACE\_TSS模板类来实现。需要成为线程专有的类被传入ACE\_TSS模板，然后可以使用C++的->操作符来调用它的全部公共方法。