ACE/C++ Network Programming Volume 1

@于龙

目录

[1 设计课题、中间件方案、ACE 3](#_Toc418527631)

[1.1 网络应用面临的挑战 3](#_Toc418527632)

[1.2 网络应用的设计空间 3](#_Toc418527633)

[1.3 面向对象中间件方案 3](#_Toc418527634)

[1.3.1 面向对象中间件层 3](#_Toc418527635)

[1.3.2 主机基础设施中间件的优势 4](#_Toc418527636)

[1.4 ACE工具包概览 4](#_Toc418527637)

[1.4.1 ACE OS Adaptation层 4](#_Toc418527638)

[1.4.2 ACE C++ Wrapper Façade层 4](#_Toc418527639)

[1.4.3 ACE framework层 4](#_Toc418527640)

[1.4.4 ACE网络服务组件层 5](#_Toc418527641)

[2 通信设计空间 5](#_Toc418527642)

[2.1 无连接协议与面向连接协议 5](#_Toc418527643)

[2.2 同步及异步消息交换 5](#_Toc418527644)

[2.3 消息传递与共享内存 5](#_Toc418527645)

[3 Socket API概述 6](#_Toc418527646)

[3.1 操作系统IPC机制概述 6](#_Toc418527647)

[3.2 Socket API 6](#_Toc418527648)

[3.3 Socket API的局限性 6](#_Toc418527649)

[4 ACE Socket Wrapper Façade 6](#_Toc418527650)

[4.1 概述 6](#_Toc418527651)

[4.2 ACE\_Addr类和ACE\_INET\_Addr类 7](#_Toc418527652)

[4.3 ACE\_IPC\_SAP 7](#_Toc418527653)

[4.4 ACE\_SOCK 7](#_Toc418527654)

[4.5 ACE\_SOCK\_Connector 7](#_Toc418527655)

[4.6 ACE\_SOCK\_IO和ACE\_SOCK\_Steam 7](#_Toc418527656)

[4.7 ACE\_SOCK\_Acceptor 8](#_Toc418527657)

[5 网络日志服务程序的实现 8](#_Toc418527658)

[5.1 ACE\_Message\_Block 8](#_Toc418527659)

[5.2 ACE\_InputCDR和ACE\_OutputCDR 8](#_Toc418527660)

[6 并发设计空间 9](#_Toc418527661)

[6.1 循环、并发及反应式服务器 9](#_Toc418527662)

[6.2 进程与线程 9](#_Toc418527663)

[6.3 进程/线程创建策略 9](#_Toc418527664)

[6.4 用户、核心及混合线程模式 9](#_Toc418527665)

[6.5 分时及实时调度级别 10](#_Toc418527666)

[7 操作系统并发机制概述 10](#_Toc418527667)

[7.1 同步事件多路分离 10](#_Toc418527668)

[7.2 多进程机制 10](#_Toc418527669)

[7.3 多线程机制 10](#_Toc418527670)

[7.3.1 互斥体 11](#_Toc418527671)

[7.3.2 Readers/Writer锁 11](#_Toc418527672)

[7.3.3 信号量锁 11](#_Toc418527673)

[7.3.4 条件变量 11](#_Toc418527674)

[8 ACE同步事件多路分离 Wrapper Facade 11](#_Toc418527675)

[8.1 ACE\_Handle\_Set 11](#_Toc418527676)

[8.2 ACE\_Handle\_Set\_Iterator 11](#_Toc418527677)

[9 ACE进程Wrapper Facade 12](#_Toc418527678)

[9.1 概述 12](#_Toc418527679)

[9.2 ACE\_Process 12](#_Toc418527680)

[9.3 ACE\_Process\_Options 12](#_Toc418527681)

[9.4 ACE\_Process\_Manager 12](#_Toc418527682)

[10 ACE线程Wrapper Facade 12](#_Toc418527683)

[10.1 概述 12](#_Toc418527684)

[10.2 ACE\_Thread\_Manager 13](#_Toc418527685)

**重点：**

设计成功的面向对象的网络应用程序和中间件需要了解的重要概念、模式和C++特性；

最常用的ACE TCP/IP及并发式wrapper façade类背后的设计动机及其基本应用.

# 设计课题、中间件方案、ACE

## 网络应用面临的挑战

在网络应用体系中，应用系统被划分为所谓的服务，供多个应用程序共享和复用.

1、了解你的应用将要提供哪些服务；

2、针对服务设计各种机制，供服务和服务、服务和客户之间通信时使用；

3、确定有哪些体系结构和服务配置能够最有效地利用现有环境；

4、利用技术和工具实现这些方案.

**固有复杂性：**

选择合适的通信机制，并设计良好的协议来有效使用这些机制；

设计合理的网络服务，有效利用资源、降低成本；

有效使用并发，获得高性能；

管理和配置服务，获得可用性和灵活性.

**偶发复杂性：**

缺乏类型安全、可移植、可扩充的原始OS API；

算法分解的广泛使用，维护和扩充困难；

核心概念和功能不断发现和创造，软件成本居高不下.

## 网络应用的设计空间

这些设计空间涉及的主要是固有复杂性的管理，改变其中一个或多个空间，网络应用的形状就会相应发生变化.

通信空间：网络应用交互的规则、形式和抽象层次；

并发空间：如何正确使用进程和线程；

服务空间：网络应用服务的主要性质，每个服务实例的持续时间和结构；

配置空间：如何识别网络服务，何时将网络服务绑定在一起，以构成完整的应用.

## 面向对象中间件方案

### 面向对象中间件层

常见的面向对象中间件体系结构，一般包括以下几个层次：

主机基础设施中间件：对OS并发、进程间通信机制进行封装，获得面向对象的网络编程能力.ACE

分布式中间件：使用主机基础设施中间件，使一些网络编程任务自动化：连接和内存管理、编解码、端点和请求的多路分离、同步、多线程.COM+、CORBA

公共中间件服务：扩展了分布式中间件，定义了更高层次、独立于领域的服务：事件通知、日志记录、持久性安全且可恢复的事务.

特定领域中间件服务：满足特定领域如电信、电子商务的特定需求.

### 主机基础设施中间件的优势

在考虑服务质量QoS和成本上的限制，主机基础设置中间件更具有优势.

满足严格的Qos需求；

控制成本；

改进核心技术，提高现有水平；

## ACE工具包概览

ACE ADAPTIVE Communication Environment 是广泛使用的基础设置中间件.

ACE设计为层次化体系结构.

ACE软件包的基础是ACE OS Adaptation层和C++ wrapper facades的结合，封装了核心OS并发网络编程机制.

ACE较高层建立在这一基础上，提供了可复用的框架、网络服务组件、基于标准的中间件等.

### ACE OS Adaptation层

适应层大约占ACE的10%.

它由一个名为ACE\_OS的类构成，此类包含500多个C++静态方法.

这些方法封装了原始的、基于C的OS API，隐藏了平台相关的细节.

### ACE C++ Wrapper Façade层

封装层占ACE的50%.

将函数和数据封装在类型安全的面向对象接口中.

### ACE framework层

面向对象框架占ACE的40%.

由多组集成在一起、相互合作的类，用来为一组相关应用提供可复用的软件架构.

事件多路分离和分发框架：Reactor和Proactor框架

连接建立和服务初始化框架：Acceptor-Connector框架，将主动和被动初始化角色同初始化结束之后，通信对等服务所执行的应用处理分离开来.

并发框架：Task框架，实现一些主要的并发模式：Active Object、Half-Sync/Half-Async.

服务配置器框架：实现Component Configuration模式.

流框架：这个框架实现了Pipes and Filters模式.

### ACE网络服务组件层

ACE还提供了一个网络服务库，库中的服务都被包装成组件.

被捆绑在ACE软件的发行包中.

演示ACE功能的常见使用方式；

提取可复用的网络应用构件.

# 通信设计空间

## 无连接协议与面向连接协议

在这一设计空间，主要权衡延迟、可伸缩性、可靠性.

无连接协议提供的是“面向消息”的服务，每条消息都独立寻址和发送.不保证次序和可靠性.

UDP、IP无连接直接被多媒体应用使用，允许一定程度的数据丢失.

面向连接协议提供的是可靠、有序、不重复的发送服务.为了提高性能并保证可靠性，面向连接协议在发送端和接收端交换并维护状态信息.

使用面向连接协议，还必须作出以下设计：

数据成帧策略：TCP是一种字节流协议，不保护消息的边界.

连接多路复用策略：多路复用、非多路复用.

## 同步及异步消息交换

请求/应答协议交换的可选策略有两种：同步和异步.

选择同步/异步策略考虑：

1. 请求之间的关联性；
2. 底层协议或传输介质的延迟.

## 消息传递与共享内存

消息传递基于IPC机制.

共享内存允许相同或不同主机上的多个进程访问、交换数据，就像数据位于每一个进程的本地地址空间.

如果数据必须被多个进程读取，共享内存是一种更有效的通信机制.

共享内存有本地和分布式两种.

本地共享内存：System V Unix共享内存、内存映射文件.

抽象类型存入共享内存将会有问题.

# Socket API概述

## 操作系统IPC机制概述

IPC分为两类：

本地IPC：共享内存、管道、门、信号等；

远程IPC：socket、命名管道.

## Socket API

最初是在BSD UNIX中开发的，用来为TCP/IP提供接口.

应用使用Socket API中的C函数来创建、管理socket套接字.

## Socket API的局限性

容易出错、过于复杂、不可移植或不统一.

# ACE Socket Wrapper Façade

这些ACE class使用了Wrapper Façade模式.

## 概述

ACE Socket Wrapper Façade带来的优势：

提高了类型安全；

保证了可移植性；

简化了常见的使用情况；

ACE Socket Wrapper Façade通过使用内联函数来保证效率.

ACE\_SOCK\_\*类封装了Internet领域的Socket API；

ACE\_LSOCK\_\*类封装了UNIX领域的Socket API；

面向连接协议中的3种角色：

主动连接角色：发起一个连接；

被动连接角色：接受一个连接；

通信角色：建立连接后交换数据。

## ACE\_Addr类和ACE\_INET\_Addr类

网络地址是Socket API中的一个薄弱点，使用了C结构和类型转换来表示.

ACE\_Addr类位于网络地址继承结构的根部.

ACE\_Addr类还定义了一个静态数据成员sap\_any，用于随机分配地址.

在每一种IPC机制中，地址类都派生自ACE\_Addr.

TCP/IP使用ACE\_INET\_Addr来表示地址.

## ACE\_IPC\_SAP

提供了ACE\_HANDLE类型定义，解决句柄可移植性问题；

定义了ACE\_INVALID\_HANDLE宏，用来检测错误.

ACE\_IPC\_SAP是ACE IPC wrapper façade继承结构的根.提供了基本的IO句柄的操作能力.

ACE\_IPC\_SAP并不提供直接使用，是一个抽象类.

## ACE\_SOCK

ACE\_SOCK是ACE Socket wrapper façade继承结构的根.

ACE\_SOCK也是抽象的.

ACE\_SOCK在析构函数中没有关闭句柄，提供了close函数，防止在复制时关闭发生.

## ACE\_SOCK\_Connector

Socket API只有数据模式和被动模式的角色，没有主动连接角色，ACE定义了ACE\_SOCK\_Connector明确定义了主动连接角色.

ACE\_SOCK\_Connector是一个工厂，用于发起一个连接，并在连接建立后初始化一个ACE\_SOCK\_Stream对象.

连接可以通过阻塞、非阻塞、定时方式发起.

## ACE\_SOCK\_IO和ACE\_SOCK\_Steam

ACE\_SOCK\_Steam明确定义了一个数据模式的只传输对象.

ACE\_SOCK\_Steam派生自ACE\_SOCK\_IO，ACE\_SOCK\_IO派生自ACE\_SOCK.

ACE\_SOCK\_Steam支持阻塞、定时、非阻塞 IO.

peer.enable(ACE\_NONBLOCK)

peer.disable(ACE\_NONBLOCK)

如果对非阻塞模式的ACE\_SOCK\_Steam调用IO方法，而且调用会阻塞，将返回-1，并将errno设为EWOULDBLOCK.

## ACE\_SOCK\_Acceptor

ACE\_SOCK\_Acceptor类是一个工厂，用来被动地建立一个新的通信端点.

ACE\_SOCK\_Acceptor使用的句柄继承自ACE\_IPC\_SAP，避免句柄误用.

# 网络日志服务程序的实现

## ACE\_Message\_Block

在消息操作中，动态内存管理的开销必须降至最低，不必要的数据复制也必须避免.

ACE\_Message\_Block类，可以高效地管理具有固定和可变长度的消息.

实现了Composite模式.

支持两种消息：

简单消息：只包含一个ACE\_Message\_Block；

复合消息：包含多个ACE\_Message\_Block，依据Composite模式连接在一起.

每一个ACE\_Message\_Block都包含一个指针，指向带引用计数的ACE\_Data\_Block.后者则指向数据的有效空间.

## ACE\_InputCDR和ACE\_OutputCDR

应用程序需要实现以下功能：

线性化；

整编/解编

CDR代表Common Data Representation格式

ACE\_InputCDR和ACE\_OutputCDR根据CDR对数据进行整编和解编.

CDR支持以下类型：

原始类型；

原始类型组成的数组；

在内部使用ACE\_Message\_Block链，避免内存复制.

利用CORBA CDR的对齐和字节次序规则

优化的字节交换代码；

# 并发设计空间

## 循环、并发及反应式服务器

在这一设计空间中，需要权衡的地方在于：是要简化编程还是要提高伸缩性.

服务器可以分为循环式、并发式和反应式等.

循环式服务器：处理后续请求前，会完整地处理每一个客户请求.

并发式服务器：同时处理多个客户请求.利用多线程或多进程.

并发式服务器的一个常见设计是一个请求一个线程.

还可以修改为一个连接一个线程.

反应式服务器：所有处理实际在一个线程中完成.多个服务请求在一个单线程进程依次循环处理.通过同步事件多路分离策略.

## 进程与线程

这一设计空间主要考虑：健壮性、效率和可伸缩性.

多线程相对于多进程的优势：

线程创建与环境切换：线程维护状态信息少，开销少.

同步：进程内同步的开销小.

数据复制：数据不需要通过内核来复制.

缺点：

性能损失：计算密集型无法受益，高精度锁策略会带来高同步开销.

健壮性降低：没有内存保护.

缺乏高精度的访问控制.

## 进程/线程创建策略

急式创建策略：形成线程或进程池.

Half-Sync/Half-Async

Leader/Followers

随需创建策略：

## 用户、核心及混合线程模式

用户线程模型：由用户空间的程序库来管理.

缺点：

无法合理使用多个CPU，进程中所有线程都竞争一个CPU.

一个线程的阻塞式操作会阻塞所有线程.

核心线程模式：内核提供了对线程的直接支持.

解决用户线程模型，但生命周期操作更昂贵.

混合线程模型：创建线程时指明这个线程属于哪种模型.

## 分时及实时调度级别

分时调度级别：

基于优先级；

公平；

抢占；

时间片；

实时调度级别：

轮流；

先进先出；

# 操作系统并发机制概述

## 同步事件多路分离

同步事件多路分离器是OS提供的一个函数，用于在一组事件源上等待特定的事件发生.当一个或多个事件源被激活时，函数将返回至调用者.

## 多进程机制

进程生存期操作；

进程同步操作；

进程属性操作；

## 多线程机制

线程生存期操作；

线程同步操作；

线程属性操作；

线程专有存储空间；

当两个或多个并发线程的执行次序造成了意想不到的错误结果时，竞态条件就会产生.

防止竞态条件的一个方法是使用同步机制，对访问共享资源的代码中的关键段实施串行访问.

常用的OS同步机制有：互斥体、多读取者/单写入者、信号量、条件变量.

### 互斥体

当共享资源被多个线程并发访问时，互斥体可确保这些资源的完整性.

互斥体可用来串行执行多个线程，这需要在代码中确定关键段.  
非递归互斥体：如果当前拥有互斥体的线程在没有首先释放它的情况下，试图再次获得它，就会导致死锁或失败.

递归互斥体：拥有互斥体的线程可以多次获得它而不产生死锁，只要这个线程最终以相同次数释放这个互斥体即可.

### Readers/Writer锁

多个线程并发读取资源，但不修改；

一次只有一个线程修改资源，其他线程此时都不能对其进行读/写访问.

可用来对读操作比写操作频繁的资源，从而提高并发程序的性能.

### 信号量锁

信号量是可以原子递增和递减的非负整数.

如果一个线程试图递减一个信号量，但这个信号量的值已经为0，则线程会阻塞.

另一个线程发出这个信号，使信号量大于0之后，被阻塞的线程才会被释放.

和互斥体不同的是，释放信号量的线程不必是最初获得这个信号量的线程.

### 条件变量

条件变量允许更复杂的调度决策.

性能消耗：非递归互斥体、递归互斥体、readers/writer、信号量、条件变量

# ACE同步事件多路分离 Wrapper Facade

## ACE\_Handle\_Set

封装select函数使用的fd\_set结构.

简化fd\_set和select在各个OS平台上的使用；

增加、删除句柄时，自动跟踪、调整fd\_set中大小相关的值.

## ACE\_Handle\_Set\_Iterator

在一个正在被遍历的ACE\_Handle\_Set中，一定不要清除句柄.

对迭代活动句柄做了大量的优化.

# ACE进程Wrapper Facade

## 概述

提高异种OS平台之间的可移植性；

易于访问和OS相关的机制；

将一组进程作为一个有机的集合来管理.

## ACE\_Process

封装了进程操作，部分操作是依赖于平台的.

## ACE\_Process\_Options

常用属性：

程序映像；

打开的IO句柄；

访问显示设备；

工作路径；

进程关系；

安全属性；

部分属性平台相关.

## ACE\_Process\_Manager

将多组进程作为一个单元来管理.

管理和监视ACE\_Process类创建的多组进程；

允许一个进程创建一组进程，并等待这一组进程退出.

# ACE线程Wrapper Facade

## 概述

确保异种OS平台间的可移植性；

将一组线程作为一个有机的集合来管理；

管理线程的调度策略和优先级；

高效地使用和管理TSS；

## ACE\_Thread\_Manager

一次创建一个线程或多个线程；

针对每一个被创建线程，改变其线程属性；

创建并管理一组线程；

管理ACE\_Task中的线程；

实现线程的协同删除；

等待一个或多个线程退出；

其删除方式不是强制，是协同式线程删除，通过ACE\_Thread\_Manager::cancel设置一个标志，指示线程应该删除自身.被删除方负责合作，周期性调用ACE\_Thread\_Manager::testcancel()检查是否被请求删除.

## ACE\_Sched\_Params

通过一种可移植的方式，指定调度策略，如先进先出、轮询；

提供某种方式，为轮询调度策略指定分时量；

提供某种方式，指定策略运用的范围；

提供一致的调度优先级表示方式；

ACE\_OS::Sched\_params()封装了用来操纵调度级别的函数.

## ACE\_TSS

支持在物理上为线程所专有，但逻辑上能像全局变量那样访问的数据；

使用C++的委托运算符operator->()来提供线程专有的智能指针；

封装了和TSS对象相关联的键的管理；

在缺乏TSS的平台上模拟TSS；