ACE/The ACE Programmer's Guide

@于龙

目录

[1 ACE简介 3](#_Toc418939716)

[1.1 ACE的组织结构 3](#_Toc418939717)

[1.2 消除C++编译器之间的差异 3](#_Toc418939718)

[2 使用ACE日志设置 4](#_Toc418939719)

[2.1 基本的日志记录和跟踪 4](#_Toc418939720)

[2.2 启用和禁用日志严重级别 4](#_Toc418939721)

[2.3 重定向日志输出 4](#_Toc418939722)

[3 收集运行时信息 4](#_Toc418939723)

[3.1 命令行参数和ACE\_Get\_Opt 4](#_Toc418939724)

[3.2 访问配置信息 5](#_Toc418939725)

[4 基本的TCP/IP Socket 用法 5](#_Toc418939726)

[4.1 一个简单的客户 5](#_Toc418939727)

[4.2 给客户增加健壮性 5](#_Toc418939728)

[5 处理事件及多个I/O流 5](#_Toc418939729)

[5.1 Reactor框架总览 5](#_Toc418939730)

[5.2 处理多个I/O源 5](#_Toc418939731)

[5.3 信号 6](#_Toc418939732)

[5.4 通知 6](#_Toc418939733)

[5.5 定时器 6](#_Toc418939734)

[5.6 Acceptor-Connector框架 6](#_Toc418939735)

[5.7 反应器实现 7](#_Toc418939736)

[6 异步I/O与ACE Proactor框架 7](#_Toc418939737)

[6.1 为何使用异步I/O 8](#_Toc418939738)

[6.2 怎样发送和接收数据 8](#_Toc418939739)

[6.3 建立连接 8](#_Toc418939740)

[6.4 ACE\_Proactor 8](#_Toc418939741)

[7 进程管理 9](#_Toc418939742)

[7.1 派生新进程 9](#_Toc418939743)

[7.2 ACE\_Process\_Manager 9](#_Toc418939744)

[7.3 ACE\_Process\_Mutex 9](#_Toc418939745)

[8 信号 9](#_Toc418939746)

[9 基本的多线程编程 9](#_Toc418939747)

[9.1 入门 9](#_Toc418939748)

[9.2 基本的线程安全性 10](#_Toc418939749)

[9.2.1 使用互斥体 10](#_Toc418939750)

[9.2.2 使用守卫 10](#_Toc418939751)

[9.3 任务间通信 10](#_Toc418939752)

[9.3.1 使用条件变量 10](#_Toc418939753)

[9.3.2 消息传递 10](#_Toc418939754)

[10 线程管理 11](#_Toc418939755)

[10.1 线程的类型 11](#_Toc418939756)

[10.1.1 调度竞争范围 11](#_Toc418939757)

[10.1.2 分离的和可会合的线程 11](#_Toc418939758)

[10.1.3 线程池 11](#_Toc418939759)

# ACE简介

## ACE的组织结构

OS适配层：提供跨平台包装函数；

wrapper façade层：类型安全、面向对象的方式封装函数和数据；

框架层：提供可供复用的架构；

网络化服务层：完整的、可复用的服务；

## 消除C++编译器之间的差异

处理编译器差异的四个主要方面：

1. 模版；
2. 数据类型；
3. 运行时初始化和关闭；
4. 分配堆内存；

运行时初始化和关闭：

ACE通过以下三个相关的类提供了可移植的解决方案：

ACE\_Object\_Manager：管理对象，可以向其登记必须销毁的对象，在关闭时会以与登记次序相反的次序销毁所有已登记的对象；

ACE\_Cleanup：ACE\_Object\_Manager使用这个类的接口管理对象生命周期.

ACE\_Singleton：实现单例模式，相比全局对象其实例化和删除都在程序控制下，还增加了线程安全性，确保只有一个对象进行初始化.

必须遵循以下两条：

1. 永远不要直接调用exit()；
2. 确保ACE\_Object\_Manager成功初始化（在不使用标准main函数时）.

分配堆内存：

统一编译器在内存分配失败时返回0或抛出异常的行为，使用宏统一返回0.

窄字符和宽字符：

ACE\_HAS\_WCHAR：启用ACE的宽字符配置

ACE\_USES\_WCHAR：指示ACE在内部使用宽字符配置

ACE\_TCHAR：根据ACE\_USES\_WCHAR使用char或者wchar\_t

ACE\_TEXT：根据ACE\_USES\_WCHAR正确地定义字符串文字

字符串类：

ACE\_CString

ACE\_WString

ACE\_TString

# 使用ACE日志设置

## 基本的日志记录和跟踪

诊断输出宏：

ACE\_DEBUG

ACE\_ERROR

ACE\_TRACE

常用严重等级：LM\_DEBUG、LM\_ERROR

ACE\_TRACE会在自己所在的地方打印一行调试信息，并在退出其作用域时打印另外一行信息.

ACE\_ASSERT

## 启用和禁用日志严重级别

ACE为每个新派生的线程自动维护该线程专有的ACE\_Log\_Msg类的单体实例,ACE\_LOG\_MSG可以快捷访问.

日志优先级分为线程级和进程级.

任意级别启用了该消息的严重级别，消息就会记入日志.

## 重定向日志输出

ACE日志默认输出到标准错误流STDERR.

可以重定向到系统日志记录器SYSLOG.

# 收集运行时信息

## 命令行参数和ACE\_Get\_Opt

ACE\_Get\_Opt可以解析两种选项：

-开头的单字符

--开头的多字符

使用：

定义短选项；

迭代；

## 访问配置信息

ACE\_Configuration\_Heap在内存中保存所有信息；

ACE\_Configuration\_Win32Registry注册表API封装

ACE\_Registry\_ImpExp含有类型信息

ACE\_Ini\_ImmpExp Windows INT格式，没有类型信息.

# 基本的TCP/IP Socket 用法

## 一个简单的客户

ACE\_INET\_Addr 基类 ACE\_Addr.

ACE\_SOCK\_Stream对象表示已连接的TCP Socket.

## 给客户增加健壮性

检查ACE\_OS::last\_error()

# 处理事件及多个I/O流

## Reactor框架总览

使用Reactor框架，应用要实现其事件处理只需做三件事：

1. 从ACE\_Event\_Handler派生一个或多个类，并实现各个虚回调方法；
2. 向ACE\_Reactor登记事件处理对象，并与感兴趣的事件关联起来；
3. 运行ACE\_Reactor事件循环.

反应器模式，基础是事件多路分离.

## 处理多个I/O源

Reactor框架的常见用途是处理来自多个来源的I/O.

反应器会把一个ACE\_Event\_Handler指针与一个句柄以及该事件处理器感兴趣的I/O事件关联在一起.

get\_handler()获得句柄的挂钩方法.

挂钩方法返回-1，handle\_close就会被调用.

## 信号

handle\_signal()

通过捕捉一个信号，执行end\_reactor\_event\_loop()来结束ACE\_Reactor循环.

## 通知

handle\_signal中的代码处在非正常的执行流中，而是处于信号状态.

在大多数平台上，如果你处在信号状态，你有许多事情不能做.

最安全的做法通常是，设置某种状态信息，把控制转到正常的执行语句.

notify -> handle\_exception()

## 定时器

handle\_timeout

被调用的时间不一定就是定时器到期的时间.

handle\_timeout接收的时间是被选择的时间，而不是当前的系统时间.

## Acceptor-Connector框架

允许程序员用模版参数指定特定的接受器类型和服务处理器类型，从而保证了框架的灵活性.

|  |
| --- |
| typedef ACE\_Acceptor<ClientService, ACE\_SOCK\_ACCEPTOR> ClientAcceptor  int ACE\_TMAIN(int, ACE\_TCHAR \*[])  {  ACE\_INET\_Addr port\_to\_listen(“HAStatus”)  ClientAcceptor acceptor;  if(acceptor.open(port\_to\_listen) == -1)  return 1;  ACE\_Reactor::instance()->run\_reactor\_event\_loop()  } |

不再继承ACE\_Event\_Handler，而继承ACE\_Svc\_Handler.

ACE\_Svc\_Handler允许你指定流类型和加锁类型.

之所以需要加锁，是因为派生自ACE\_Task，后者含有一个ACE\_Message\_Queue成员，你必须为这个成员提供同步类型.

ACE\_Svc\_Handler::open()的默认行为就是针对READ事件登记处理器.返回-1会调用close()挂钩方法.

ACE\_Connector与ACE\_Acceptor一样，产生一个ACE\_Svc\_Handler派生的对象，用于在连接建立后运行服务.

## 反应器实现

ACE通过使用桥接模式，可以改变反应器的实现.

改变实现：

|  |
| --- |
| ACE\_TP\_Reactor \*tp\_reactor = new ACE\_TP\_Reactor;  ACE\_Reactor \*my = new ACE\_Reactor(tp\_reactor, 1);  ACE\_Reactor::instance(my\_reactor,1) |

ACE\_Select\_Reactor：除Windows之外的所有平台的默认实现；

ACE\_WFMO\_Reactor：Windows上的默认实现；  
限制：

只能登记62个句柄；

只支持handle\_input、handle\_output、handle\_exception回调；

支持多个线程；

延迟的处理器移除.

如果使用的是COM/DCOM服务器，应该使用ACE\_Msg\_WFMO\_Reactor.

ACE\_TP\_Reactor：扩展了ACE\_Select\_Reactor，允许同时在多个线程中运行.

ACE\_Priority\_Reactor：根据ACE\_Event\_Handler的prioprity()的优先级分派.

# 异步I/O与ACE Proactor框架

必须在多个端点上进行I/O操作的应用会使用以下两种I/O模型之一：

1、反应式；

2、多线程，伸缩性不是很好；

异步I/O也称为前摄式I/O的伸缩性常常更好.

前摄式I/O允许应用在多个I/O端点上并行地发起一个或多个I/O请求，并且不用为等待它们的完成而阻塞.当每个操作完成时，OS会通知某个完成处理器，由它随后对结果进行处理.

## 为何使用异步I/O

反应式I/O常常在单线程中进行，由反应器的事件分派循环驱动.但在同一时刻每个线程只能执行一个I/O操作，这种本质可能会成为瓶颈，因为在多个端点上传输大量数据的应用无法利用OS或多个CPU或网络接口的并行能力.

使用多线程的缺点：

线程policy与并发policy紧密地耦合；

增加了同步复杂性；

同步造成性能的退化；

如果只是为了提供I/O并行度，使用多线程并非总是好的选择；

前摄式I/O模型有两个不同的步骤：

发起I/O操作；

在后面的某一时刻处理该操作的完成；

允许单线程化应用并发地执行多个I/O操作，并且不会带来与传统的多线程化机制相关的开销和复杂性；

## 怎样发送和接收数据

ACE\_Service\_Handler创建新服务处理器所用的目标类.

ACE\_Handler是ACE\_Service\_Handler父类，定义了处理异步I/O完成所需的接口.

ACE\_Asynch\_Read\_Stream/ACE\_Asynch\_Write\_Steam读写I/O的工厂类.

Proactor与Reactor框架不同，没有显示的处理器.Proactor与完成处理器对象之间的唯一联系就是未完成的IO操作.

## 建立连接

ACE\_Asynch\_Acceptor用于发起被动的连接建立；

ACE\_Asynch\_Connector用于发起主动的连接建立.

## ACE\_Proactor

ACE\_Proactor类负责驱动ACE Proactor框架的完成处理.

这个类等待完成事件的发生、把这些事件多路分离给相关的完成处理器，并分派每个完成处理器上适当的挂钩方法.

各个操作系统上的异步I/O设施有很大的不同，通过使用桥接模式来维持灵活性和扩展性.

ACE\_WIN32\_Proactor

ACE\_POSIX\_Proactor

# 进程管理

## 派生新进程

ACE\_Process包装类，通常会为每个新进程使用一个ACE\_Process对象.

ACE\_Process\_Options设置：

设置标准I/O句柄；

指定两个进程间句柄继承的工作方式；

设置子进程的环境和命令行；

指定安全属性；

## ACE\_Process\_Manager

进程管理器允许单次调用派生多个进程，并等待它们的终止.

可以结合Reactor框架来登记事件处理器，获得进程退出事件.

## ACE\_Process\_Mutex

ACE以ACE\_Process\_Mutex类的形式提供了有名互斥体，可以跨越多个地址空间使用.

因为互斥体是有名称的，通过把相同的名称传入ACE\_Process\_Mutex构造器，你可以重新创建代表同一个互斥体的对象.

# 信号

信号充当的是软件中断，指示异步事件的发生.

# 基本的多线程编程

## 入门

要通过ACE创建你自己的线程，你所需做的只是创建ACE\_Task\_Base类的子类，并重新定义svc虚方法的实现.

## 基本的线程安全性

### 使用互斥体

acquire()

release()

ACE\_Thread\_Mutex

### 使用守卫

守卫基于一种常见的C++惯用手法：把构造器和析构器用于资源的获取和释放.

可以使用宏或类.

## 任务间通信

任务间通信一般可以划分为两个宽泛的范畴：

状态变化或事件通知；

消息传递；

### 使用条件变量

线程可以用条件变量来把状态的变化、事件的达到或是另外的条件的满足传达给其他感兴趣的线程.条件变量总是要与互斥体协同使用.

条件变量还有一个特殊的特征：你可以在其上进行定时的阻塞.

条件变量总是与互斥体协同使用

### 消息传递

从ACE\_Task派生的类，就自动继承了一个ACE\_Message\_Queur类型的消息队列.

多种队列类型.

# 线程管理

## 线程的类型

可以把各种线程类型的标志传递给activate()

### 调度竞争范围

内核级线程是OS通过内核级的调度器对它们进行调度.

用户级线程是基于库的调度器在进程的地址空间中进行调度.

THR\_BOUND标志会显示地把某个用户级线程绑定到一个内核级线程，这样调度就会使用底层的内核级线程来完成.

THR\_NEW\_LWP会创建一个新的内核级可调度实体.

### 分离的和可会合的线程

线程指定为分离的，ACE会在线程退出时自动清理这个线程所持有的资源.但其他线程无法查看这个线程的退出状态.

线程如果是可会合的，另外的线程就可以与之会合，如果你没有进行会合，就会泄露资源.

### 线程池

activate方法允许在同一个时刻启动多个线程.

grp\_id()获得线程组的标示符.

创建的所有线程共享同一个ACE\_Task，但拥有自己独立的栈.