ACE 概述

Allen Long
ihuihoo@gmail.com
http://www.huihoo.com
Huihoo - Enterprise Open Source

内容安排

- ACE介绍
  - ACE的获得和安装
  - ACE综述
  - ACE架构层次
  - ACE OS适配层
  - ACE OO包装
- GoF模式：

Strategy,Singleton,Bridge,Facade,Composite 的介绍

内容安排

- ACE基础
- ACE模式

Reactor,Proactor,Component Configurator,

ActiveObject,Half-Sync/Half-Async,Acceptor-onnector,Pipes and Filters

- 使用ACE的日志服务
- 收集运行时信息
- ACE容器
- ACE成功应用

ACE是什么？

- ACE自适配通信环境（ADAPTIVE Communication Environment）是可以自由使用、开放源码的面向对象（OO）框架（Framework），在其中实现了许多用于并发通信软件的核心模式。ACE提供了一组丰富的可复用C++ Wrapper Facade（包装外观）和框架组件，可跨越多种平台完成通用的通信软件任务，其中包括：事件多路分离和事件处理器分派、信号处理、服务初始化、进程间通信、共享内存管理、消息路由、分布式服务动态（重）配置、并发执行和同步，等等。
- ACE的目标用户是高性能和实时通信服务和应用的开发者。它简化了使用进程间通信、事件多路分离、显式动态链接和并发的OO网络应用和服务的开发。此外，通过服务在运行时与应用的动态链接，ACE还使系统的配置和重配置得以自动化。
- ACE可看作与Java Virtual Machine (JVM) 和Microsoft Common Language Runtime (CLR)处在一个层次的基础中间件,
,在其上可构建分布式中间件等专属中间件:如TAO

ACE的层次

(1) ACE OS Adaptive 层

* 多线程和同步
* 进程间通信
* 事件分离
* 直接动态链接
* 内存映射文件和共享内存

(2) ACE C++ wrapper Facade 层

* IPC-SAP
* 访问初始化 -- Connector 和 Acceptor 组件
* 并发机制 -- 主动对象
* 内存管理机制
* CORBA 集成

ACE的层次

(3) ACE framework 层

* 事件多路分离和分发框架: Reactor, Proactor 框架分别实现了 Reactor 模式, Proactor 模式. Reactor 和 Proactor 框架自动处理"和应用相关" 的处理程序(handler)的多路分离和分发, 以响应各种基于 I/O, 计时器, 信号和同步的事件.
* 连接建立和服务初始化框架: Acceptor-Connector 框架实现了 Acceptor-Connector 模式. 这个框架将 "主动和被动初始化角色" 同 "初始化结束后, 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, 如 Active Object, Half-Sync/Half-Asynsc
* 服务配置器框架: Service Configurator 实现了 component configuration(组件配置器)模式, 以支持应用程序的配置.
* 流(Streams)框架: 这个框架实现了 Pipes and Filters (管道和过滤器)模式. 对于那些可以灵活地组合起来, 从而创建某种网络应用(譬如, 用户级协议栈和网络管理代理等)并具有层次化的结构的服务来说, ACE 的 "流" 框架可以简化其开发.

(4) 网络服务组件层

- 演示 "ACE 功能的常用方式"
- 提取 "可复用的网络应用组件"

* 名称管理
* 事件路由处理
* 日志记录
* 时间同步
* 网络锁定

ACE能做什么？

ACE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:

- 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。

获得,安装ACE (Linux)

- http://deuce.doc.wustl.edu/Download.html
- Linux安装

download ACE-5.7.zip
将包解压缩到 /usr/local/ACE_wrappers
设置环境变量:

```
export ACE_ROOT=/usr/local/ACE_wrappers
export TAO_ROOT=$ACE_ROOT/TAO
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACE_ROOT/ace
```

你可以将这些设置到/etc/profile ,这样Linux启动后,这些环境就建立好啦

1. 设置编译所需文件

(1)$ACE_ROOT/ace/config-linux.h to copy to $ACE_ROOT/ace/config.h,
(2)a $ACE_ROOT/include/makeinclude/platform_linux.GNU to make a symbolic link for ACE_ROOT/include/makeinclude/platform_macros.GNU.

2. 在$ACE_ROOT/ace/目录下执行 make

经过很长一段编译时间后,会在:$ACE_ROOT/ace/目录下产生很多so文件,包括:libACE.so

3. 编译ACE examples

```
cd /usr/local/ACE_wrappers/examples
make
```

这时你就可以测试ACE带的例子啦. (TAO也可用类似的方法继续编译)

获得,安装ACE (Linux)

Linux安装

1、设置设置ACE_ROOT环境

vi /etc/profile

在其中加入以下内容

```
ACE_ROOT=/opt/ACE export ACE_ROOT
LD_LIBRARY_PATH=$ACE_ROOT/ace:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

2、从包中解压出ACE-install.sh

```
tar －zxvf ACE-5.4+TAO-1.4.tar.gz ACE_wrappers/ACE-install.sh
mv /ACE_wrappers/ACE-install.sh .
cp ACE-install.sh ACE-install.sh.bak
vi ACE-install.sh
```

--MY_ACEDIR=${HOME}/aceconfig改为MY_ACEDIR=/home/my/ACE

--MY_ACE_CONFIG=config-sunos5.4-sunc++-4.x.h改为MY_ACE_CONFIG=config-linux.h

--MY_ACE_GNU_MACROS=platform_sunos5_sunc++.GNU改为
MY_ACE_GNU_MACROS=platform_linux.GNU

3、./ACE-install.sh

获得,安装ACE (Windows)

- http://deuce.doc.wustl.edu/Download.html
- Windows安装

download ACE-5.7.zip

将包解压缩到 C:\ACE_wrappers

  – 设置ACE_ROOT=C:\ACE_Wrappers

  - 在C:\ACE_Wrappers\ace下建立一个config.h文件,

加入

```
#include "config-win32.h"
#define ACE_USE_WCHAR
#define ACE_HAS_WCHAR // 支持unicode
```

  - 在VS2005或VS2008 中打开ACE_vc8.sln, ACE_vc9.sln 开始编译

设置classpath=c:\ace_wrappers\bin 使程序能找到aced.lib等相关包

选择安装

- 根据需要选择安装不同的组件

download ACE-5.7.zip
将包解压缩到 C:\ACE_wrappers
  – 设置ACE_ROOT=C:\ACE_Wrappers
  - 在C:\ACE_Wrappers\ace下建立一个config.h文件,
加入
#include “config-win32.h”
#define ACE_USE_WCHAR
#define ACE_HAS_WCHAR // 支持unicode
  - 在VS2005或VS2008 中打开ACE_vc8.sln, ACE_vc9.sln 开始编译
设置classpath=c:\ace_wrappers\bin 使程序能找到aced.lib等相关包

集成ACE与VS2005, VS2008

一.打开 Tools>Options>Projects and Solutions
加入以下内容:
Executable File:
C:\ACE_wrappers\bin

Include File:
C:\ACE_wrappers\
C:\ACE_wrappers\TAO
C:\ACE_wrappers\TAO\orbsvcs

Library Files:
C:\ACE_wrappers\lib
C:\ACE_wrappers\TAO\tao
C:\ACE_wrappers\TAO\orbsvcs\orbsvcs

Source Files:
C:\ACE_wrappers\ace
C:\ACE_wrappers\TAO\tao
C:\ACE_wrappers\TAO\orbsvcs\orbsvcs

二.加入 C:\ACE_wrappers\bin和C:\ACE_wrappers\lib加到系统环境变量 path中

三.设置系统变量
ACE_ROOT=C:\ACE_wrappers
TAO_ROOT=C:\ACE_wrappers\TAO

将一个带有IDL文件的dsw项目装入VC6
然后在工具条>工程>设置>点左边IDL文件,此时右边会出现设置项
C命令
..\..\..\..\bin\tao_idl -Ge 1 -GC $(InputName),idl

O输出
$(InputName)C.h
$(InputName)C.cpp
$(InputName)C.i
$(InputName)S.h
$(InputName)S.i
$(InputName)S.cpp
$(InputName)S_T.h
$(InputName)S_T.cpp
$(InputName)S_T.I

或在命令行下编译cd %ACE_ROOT%\bin\
tao_idl <option> IDL-file(s)

tao_idl -u or tao_idl -? 获得帮助

运行examples,测试编译安装

第一个ACE应用

```
#include “ace/Log_Msg.h”

int ACE_TMAIN(int, ACE_TCHAR *[])

{
    ACE_DEBUG((LM_INFO, ACE_TEXT("Hello ACE\n")));
   return 0;
}
```

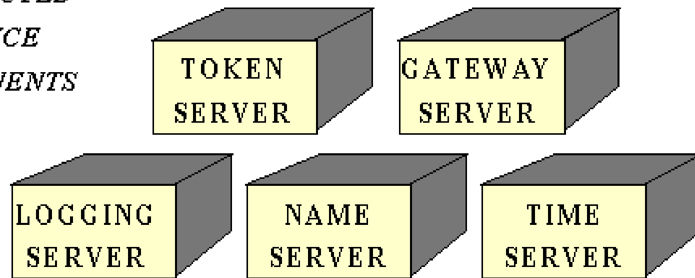
Windows:

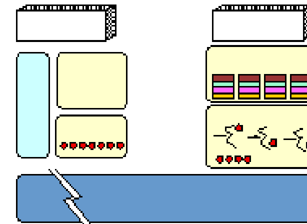
```
cl hello_ace.cpp /I "C:\ACE_Wrappers" /DWIN32 /link
    "C:\ACE_Wrappers\lib\ACEd.lib"
```

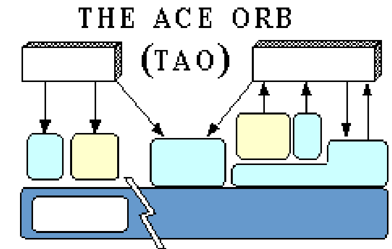
Linux:

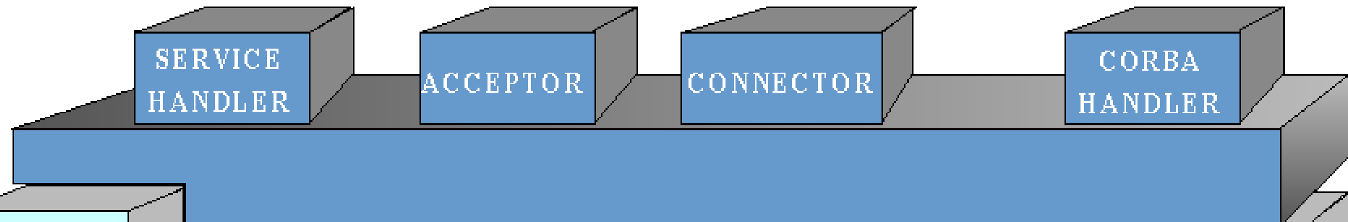
```
gcc -lACE -I"/home/ACE_wrappers" -L"/home/ACE_wrappers/lib" -o
    hello_ace hello_ace.cpp
```

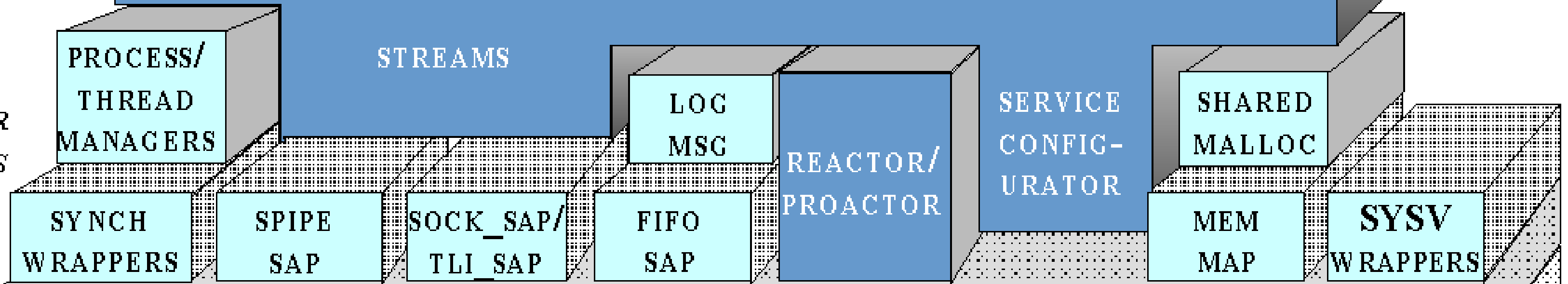
ACE中的关键组件以及它们的层次关系

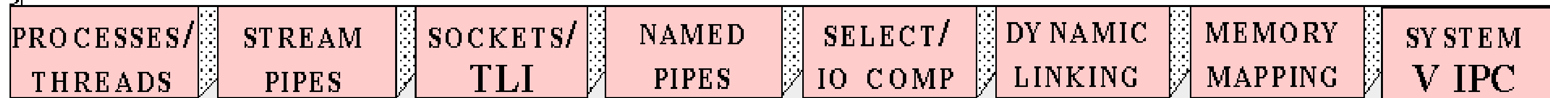




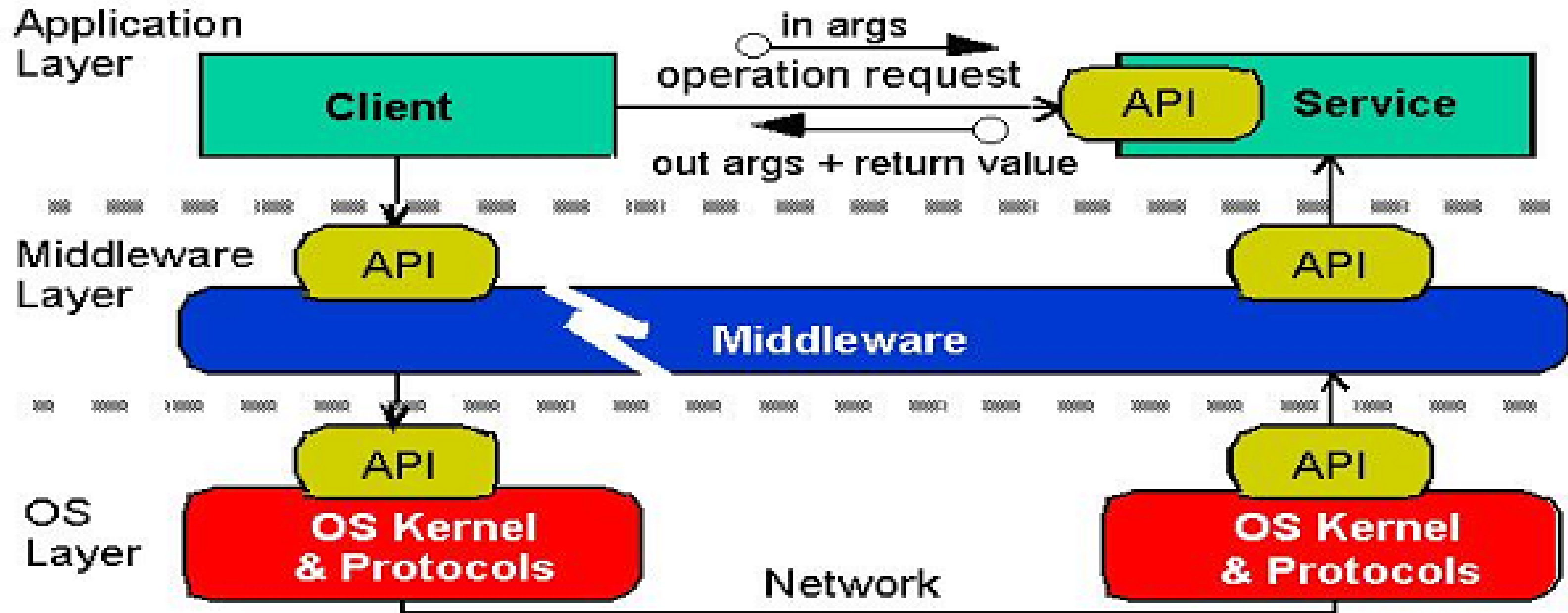








ACE提供的核心能力一



服务的访问和控制(三层结构)

服务访问和配置模式

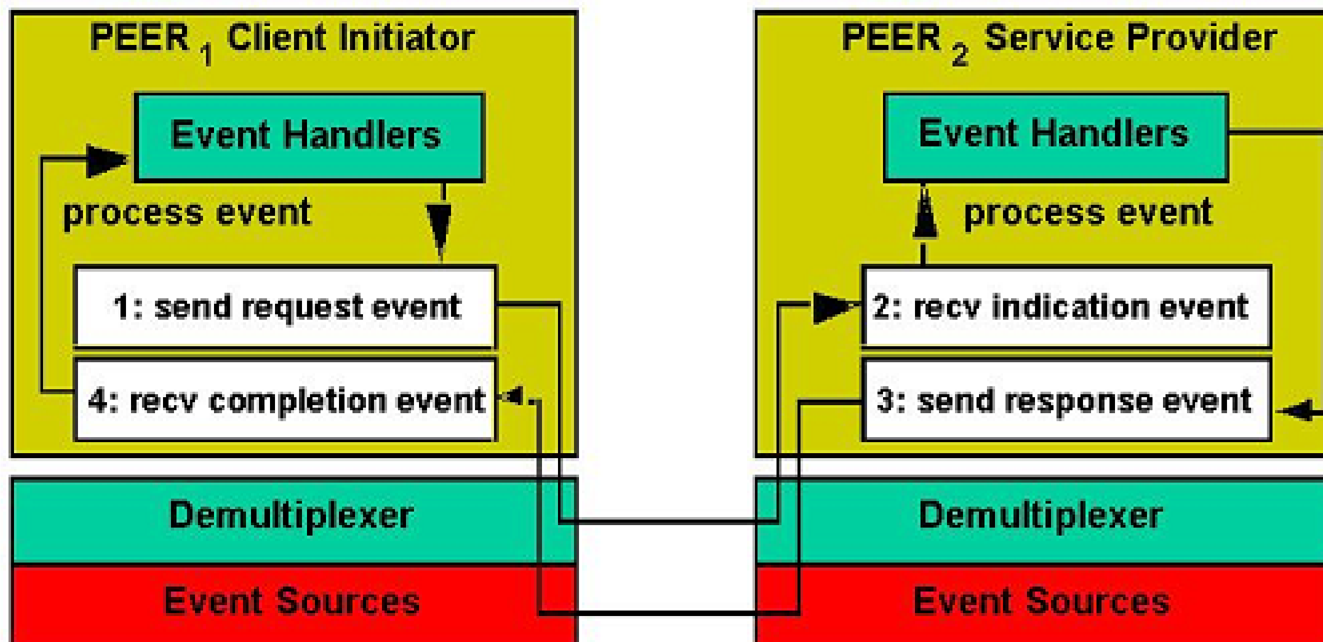
-- **包装器外观(Wrapper Facade)**
把现有的非面向对象的API所提供的函数和数据，封装在更简洁、更健壮、可移植、可维护的内聚的面向对象的类接口中

-- **组件配置器(Component Configurator)**
允许应用不必修改和重新编译的情况下，在运行时连接和解除连接其组件实现。支持不关闭和重起服务进程的情况下，将组件配置进不同的进程中

-- **拦截器(Interceptor)**
允许透明地把服务加到框架中,并且在某些事件发生时,能自动地触发服务

-- **扩展接口(Extension Interface)**
允许组件导出多个接口,当开发人员扩展或修改组件的功能时，此模式能防止接口的膨胀和客户机代码的破坏

ACE提供的核心能力二



事件处理和IPC

事件处理模式

-- 反应堆(Reactor)
使事件驱动的应用可以多路分解和分发从客户端的请求

-- 前摄器(Proactor)
使事件驱动的应用可以多路分解和分发由异步操作所激发的服务请求

-- 异步完成标记(Asynchronous Completion Token,ACT)
可以使应用程序能有效地多路分解和处理对调用服务的异步操作的响应

-- 接受器-连接器(Acceptor-Connector)
将网络系统中对等服务的连接和初始化工作，与该服务在连接和初始化后要执行的处理划分开

Reactor是ACE事件分发的基础

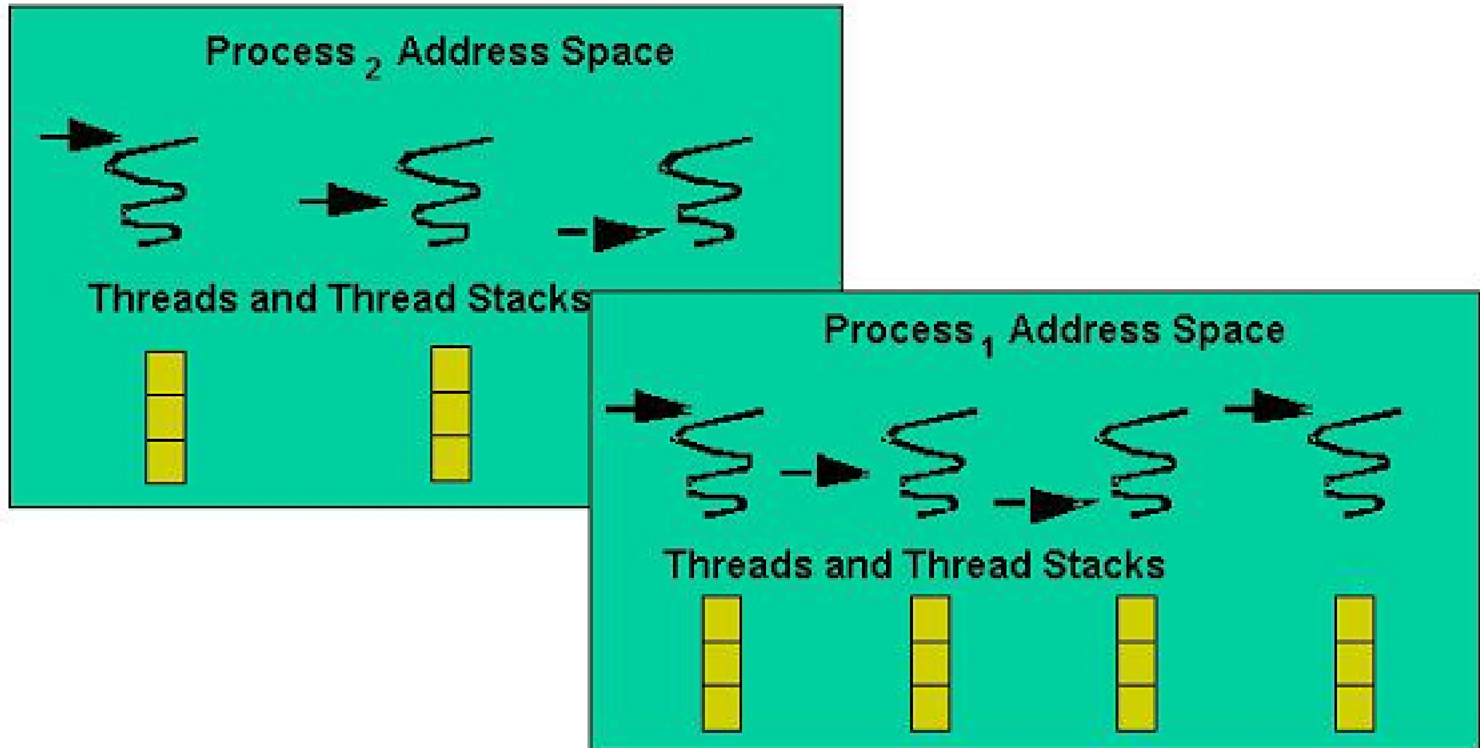
反应堆(Reactor)模式为高效的事件多路分离和分派提供可扩展的面向对象构架.目前用于事件多路分离的OS抽象既复杂又难以使用，因而也容易出错. 反应堆本质上提供一组更高级的编程抽象，简化了事件驱动的分布式应用的设计和实现. 除此以外，反应堆还将若干不同种类的事件的多路分离集成到易于使用的API中. 特别地，反应堆对基于定时器的事件、信号事件、基于I/O端口监控的事件和用户定义的通知进行统一地处理.

ACE_wrappers\examples\Reactor目录下

用Reactor创建事件分发器

```
#include "ace/Reactor.h"
#include "ace/Event_Handler.h"
#define  SIGWINCH 28
#define  SIGINT 2 //按下control + c
class MyEventHandler: public ACE_Event_Handler
{
	int handle_signal(int signum, siginfo_t*,ucontext_t*) //处理信号
	{
	   switch(signum)
	   {
	      case SIGWINCH: ACE_DEBUG((LM_DEBUG, "You pressed SIGWINCH \n"));
	      break;
	      case SIGINT: ACE_DEBUG((LM_DEBUG, "You pressed SIGINT \n"));
	      break;
	   }
	   return 0;
	}
};
int main(int argc, char *argv[])
{
	MyEventHandler *eh =new MyEventHandler;
	ACE_Reactor::instance()->register_handler(SIGWINCH,eh); //注册信号1
	ACE_Reactor::instance()->register_handler(SIGINT,eh); //注册信号2
	while(1)
	   ACE_Reactor::instance()->handle_events();
}
```

ACE提供的核心能力三



并发控制

并发模式

-- 主动对象(Active Object)

将方法执行和方法调用分离，加强并发和简化对驻留在自身控制线程中对象的同步访问. 它给线程间的协作处理提供了基于对象的解决方案.(6个参与者)

-- 监视器对象(Monitor Object)

使并发访问的执行同步化，以确保任一时刻仅有一个方法在对象内运行. 它也允许对象方法相互协调，调度方法的执行顺序. 也叫线程安全的被动对象. (4个参与者)

-- 半同步/半异步(Half-Sync/Half-Async)

将并发系统中异步和同步服务处理分离，简化了编程，同时又没有降低性能. 有两个通信层，一个用于异步服务处理，一个用于同步服务处理.

-- 领导者/追随者(Leader/Followers)

为了检测、多路分解、分发和处理事件源上引发的服务请求，多线程轮流共享一个事件源集合.

-- 线程专有存储(Thread-Specific Storage)

允许多个线程使用一个“逻辑上全局”的访问点获得一个局限于某一线程的对象，而不会导致对象访问中的加锁开销.

ACE_Task是ACE并发框架的基础

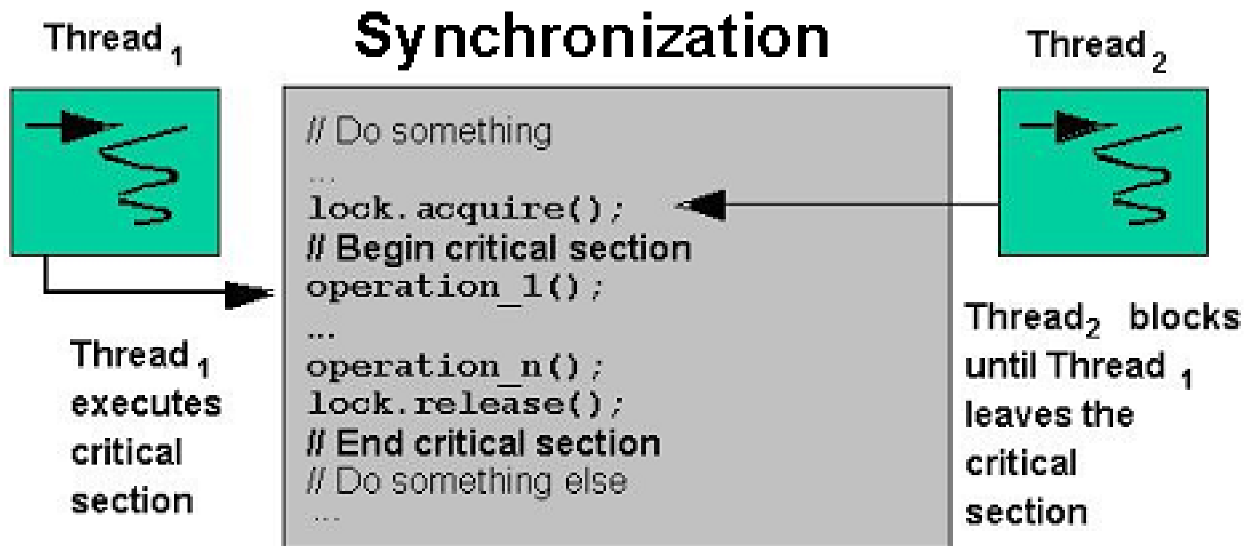
- ACE_Task类将多线程和面向对象的编程和队列集成在一起
- 它使用ACE_Message_Queue的一个实例来使数据和请求与其处理相分离，从ACE_Task派生的类， 会自动继承一个ACE_Message_Queue类型的消息队列；
- 它使用ACE_Thread_Manager来激活任务
- 它继承自ACE_Service_Object,所以可以通过ACE Service Configurator框架对它进行动态配置
- 它是ACE_Event_Handler的后代，所以它的实例可以在ACE Reactor框架中充当事件处理器
- 它提供多个虚构子方法，应用类可以为任务特有的服务执行和消息处理而重新实现它们。

ACE_wrappers\examples\Threads目录下

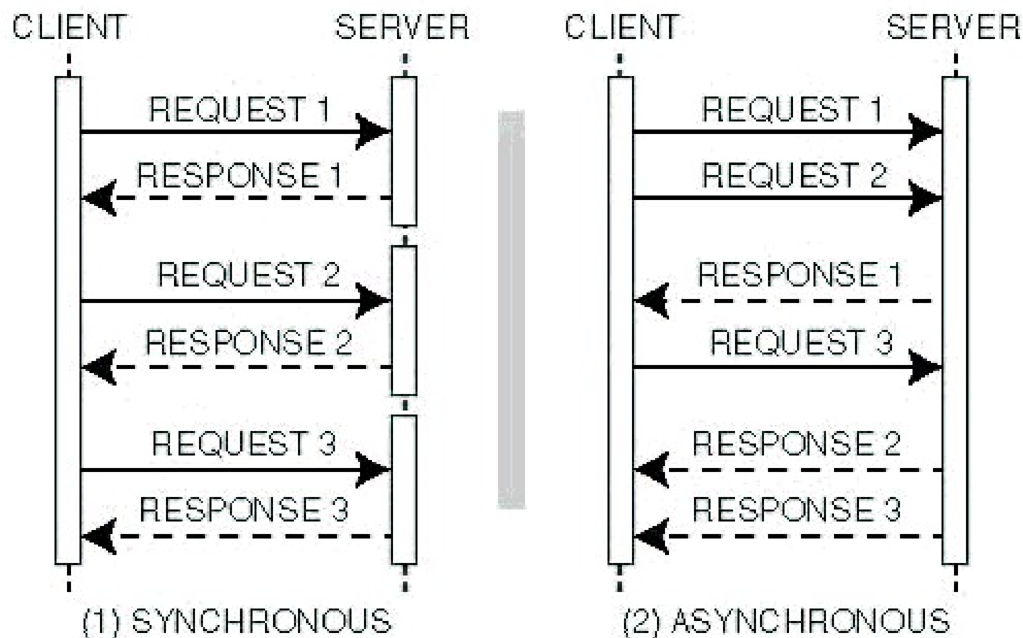
用Task创建线程

```
#include "ace/Task.h"
#include "ace/Log_Msg.h"

class Task_Thread : public ACE_Task<ACE_MT_SYNCH>
{
public:
  virtual int svc (void)
  {
    ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("(%t) starting up \n")));
    ACE_Message_Block *mb;
    if (this->getq (mb) == -1)
          { return -1;     }
    return 0;
  }
};

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  Task_Thread handler;

  handler.activate (THR_NEW_LWP | THR_JOINABLE, 4);
  handler.wait ();
  return 0;
}
```

ACE提供的核心能力四



同步处理



同步的消息交换

-Requests & responses are exchanged in a lock-step sequence.

-Each request must receive a response synchronously before the next is sent

同步模式

-- 范围锁(Scoped Locking)
确保当控制进入到某一个范围时，自动获得锁，当控制离开该范围时，自动释放锁.

-- 策略锁(Strategized Locking)
把同步机制参数化，保护组件的临界区免受并发访问.

-- 线程安全接口(Thread-Safe Interface)
将加锁开销减至最少，并保证组件间的方法调用不会因为想再次获得一个已被组件拥有的锁而导致的“自死锁”.

-- 双检查加锁优化(Double-Checked Locking Optimization)
如果代码的临界区必须在程序执行内只以线程安全的方式获得一次锁时，该模式能够减少争用和同步开销.

ACE中的同步保护原语

原语	描述
ACE_Mutex	封装互斥机制的包装类, 用于为共享资源的序列化访问提供简单、高效的机制. 其功能与二元信号量类似. 可用于线程和进程的互斥.
ACE_Thread_Mutex	可用于替换ACE_Mutex, 专用于线程同步.
ACE_Recursive_Thread_Mutex	递归的互斥体, 可以由同一线程多次获取. 但必须释放的次数和获取的次数一样多.
ACE_RW_Mutex	封装Reader/Writer锁的包装类. 要进行读和写, 要以不同的方式获取这种锁, 在没有写入者时可以有多个读取者进行读取. 锁是非递归的
ACE_RW_Thread_Mutex	可用于替换ACE_RW_Mutex, 专用于线程同步.
ACE_Token	令牌是ACE提供的最丰富、最重的加锁原语. 这种锁是递归的, 允许你进行读加锁和写加锁. 此外, 它还会保证严格的FIFO获取顺序. 所有调用acquire()的线程都会进入一个FIFO队列, 按序获取锁.
ACE_Atomic_Op	模板包装, 允许你在指定的类型上进行安全的算术运算
ACE_Guard	基于模板的守卫类, 其模板参数是锁的类型
ACE_Read_Guard ACE_Write_Guard	守卫类, 会在构造过程中调用一个读/写守卫的acquire_read()或acquire_write()

ACE线程安全与同步例子

```
#include "ace/Thread.h"
#include "ace/Thread_Mutex.h"
#include <iostream>
using namespace std;

ACE_Thread_Mutex mutex;
void* Thread1(void *arg)
{
   mutex.acquire();
   ACE_OS::sleep(3);
   cout<<endl<<"hello thread1"<<endl;
   mutex.release(); // 若没释放, 则另一个线程无法获得
   return NULL;
}
void* Thread2(void *arg)
{...}

int main(int argc, char *argv[])
{
     ACE_Thread::spawn((ACE_THR_FUNC)Thread1);
     ACE_Thread::spawn((ACE_THR_FUNC)Thread2);
     while(true)
        ACE_OS::sleep(3);
     return 0;
}
```

OS适配层

ACE_OS()

ACE OS适配层的可移植性使得ACE可运行在许多操作系统上：

- 该层位于用C写成的本地OS API之上
- 由于ACE的OS适配层所提供的抽象机制，所有OS平台使用同一棵代码树,这样的设计极大地增强了ACE的可移植性和可维护性。
- ACE屏蔽了以下的一些与OS API相关联的平台属性:

1. 并发和同步
2. 进程间通信（IPC）和共享内存
3. 事件多路分离机制
4. 显式动态链接
5. 文件系统机制

C++包装层

--ACE C++包装了IPC、服务初始化、并发、内存管理等机制

--ACE采用C++类和对象、而不是独立的C函数来包装这些机制

--ACE使用了C++的语言特性:比如模板、继承和动态绑定

--C++的使用提高了应用的健壮性,(C++包装是强类型的,这样避免了用户直接访问弱类型C接口编写的底层OS库）

--ACE大量地使用C++内联来消除额外的方法调用开销

--由于采用了OO的编程方式,带来了对模块性和可扩展性的强调,将易变的实现细节封装在稳定的接口后面,这样增强了软件的复用。

...

ACE的主要做法

ACE能够帮助你处理或绕开编译器差异的四个主要方面：

-- 模板，模板的使用和初始化

-- 数据类型，即是硬件的，也是编译器的

-- 运行时初始化和关闭

-- 分配堆内存

模板

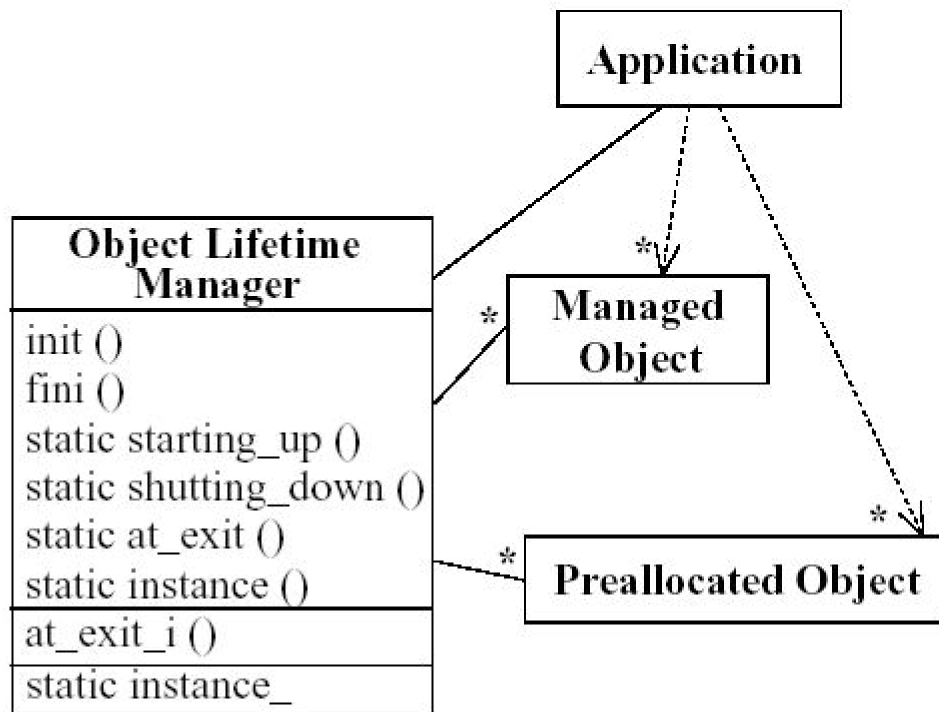
ACE使用了大量的模板技术. 模板可衍生出一系列的类和函数,是源码级的复用,而继承是目标代码的复用.

```
//TYPE_T必须是能做小于运算且返回布尔值的类型
template<class TYPE_T>
void mySort (TYPE_T[] items,int itemNum)
{
  TYPE_T* pMinItme,pCurItem;
//…
If (*pMinItem < *pCurItem)
//…
}
```

与字符宽度有关的宏

宏	用途
ACE_HAS_WCHAR	用于启用ACE的宽字符方法的配置设置
ACE_USES_WCHAR	用于指示ACE在内部使用宽字符的配置设置
ACE_TCHAR	匹配ACE的内部字符宽度; 取决于ACE_USES_WCHAR是否设置, 或者定义为 char, 或者定义为wchar_t
ACE_TMAIN	基于ACE_USES_WCHAR, 适当地为命令行参数类型定义程序的主入口点
ACE_TEXT(str)	基于ACE_USES_WCHAR, 正确的定义字符串文字
ACE_TEXT_CHAR_TO_TCHAR(str)	如果需要, 把 char * 字符串转换为 ACE_TCHAR 格式
ACE_TEXT_WCHAR_TO_TCHAR(str)	如果需要, 把 wchar_t * 字符串转换为 ACE_TCHAR 格式
ACE_TEXT_ALWAYS_CHAR(str)	如果需要, 把ACE_TCHAR 字符串转换为 char * 格式

对象管理



1、被管理对象：任何一个向对象生命周期管理者注册并由其负责销毁的对象.对象销毁发生在对象生命周期管理者本身被销毁的时候,通常都是在程序中止的时候.

2、预分配对象：被对象生命周期管理者在其内部通过硬编码方式实现创建和销毁的对象.它和对象生命周期管理者具有相同的生命周期,也就是执行应用的进程的生命周期.

3、应用：应用清晰或非清晰的创建和销毁对象生命周期管理者.此外,应用向本身可能包含预分配对象的对象生命周期管理者注册被管理对象.

对象管理

1. ACE_Object_Manager : 对象管理
   ACE::init() 初始化对象管理器
   ACE::fini() 关闭对象管理器

2. ACE_Cleanup : ACE_Object_Manager使用它管理对象生命周期,每个通过对象管理器注册的对象都派生自ACE_Cleanup

3. ACE_Singleton : 创建对象实例

注意与ACE Service Configurator框架中ACE_Service_Object类的区别

堆内存分配

-- C语言编写

```
char *c = (char *)malloc (64);              // 函数
if (c == 0) // failure
exit(1);
```

-- C++语言编写

```
char *c = new char[64];                      // 运算符
if (c == 0) // failure
exit(1);
```

-- ACE编写

```
char *c;
ACE_NEW_NORETURN (c, char[64]);  // 宏
if (c == 0) // failure
exit(1);
```

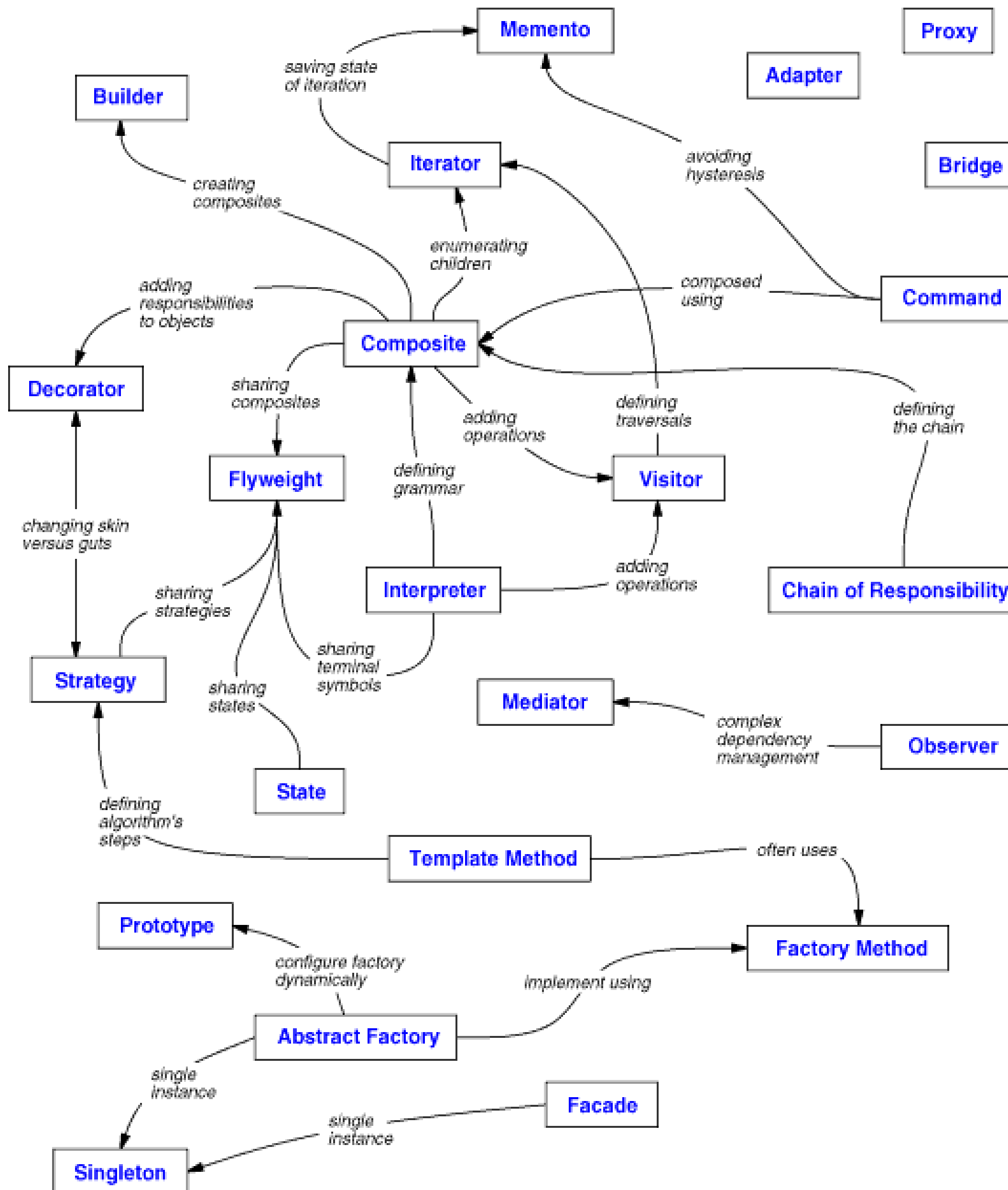
ACE内存分配宏

宏	动作
ACE_NEW(p, c)	使用构造器c分配内存, 并把指针赋给p. 在失败时, p被设置为0并执行return
ACE_NEW_RETURN(p, c, r)	使用构造器c分配内存, 并把指针赋给p. 在失败时, p被设置为0并执行return r
ACE_NEW_NORETURN(p, c)	使用构造器c分配内存, 并把指针赋给p. 在失败时, p被设置为0并继续执行下一条语句

ACE架构组件

ACE架构组件是ACE中最高层次的可用部件，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架构组件来帮助自己在很高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“微型体系结构”，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非常有用的。ACE的这一层含有以下一些架构组件：

GoF 23种模式



创建模式

- Factory(工厂方法和抽象工厂)
- Prototype(原型)
- Builder
- Singleton(单态)

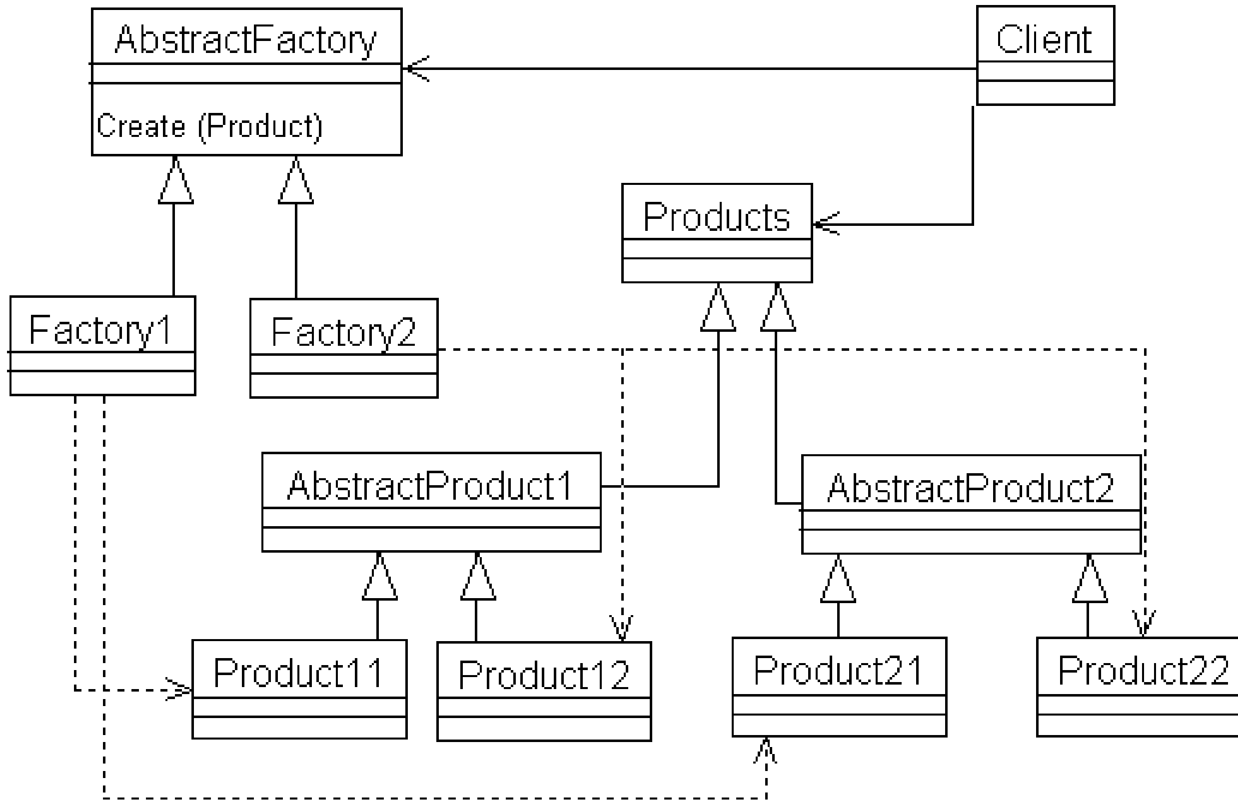
结构模式

- Facade
- Proxy
- Adapter
- Composite
- Decorator
- Bridge
- Flyweight

行为模式

- Template
- Memento
- Observer
- Chain of Responsibility
- Command
- State
- Strategy
- Mediator
- Interpreter
- Visitor
- Iterator

GoF Abstract Factory模式

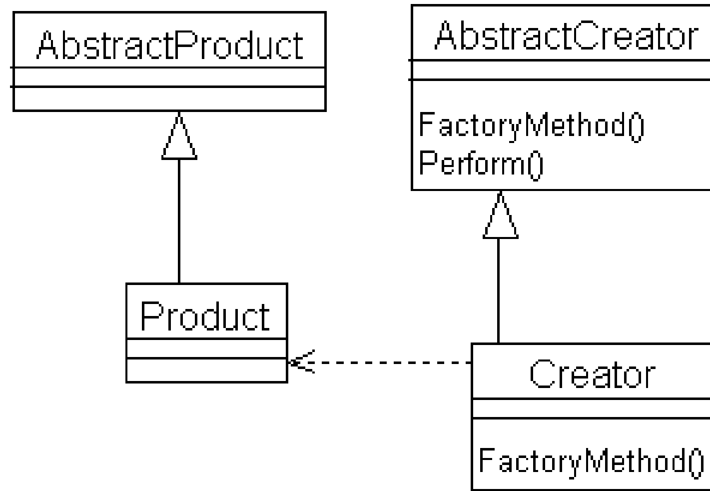


。工厂模式就相当于创建实例对象的new，我们经常要根据类Class生成实例对象，如A a=new A() 工厂模式也是用来创建实例对象的

。两个模式区别在于需要创建对象的复杂程度上。

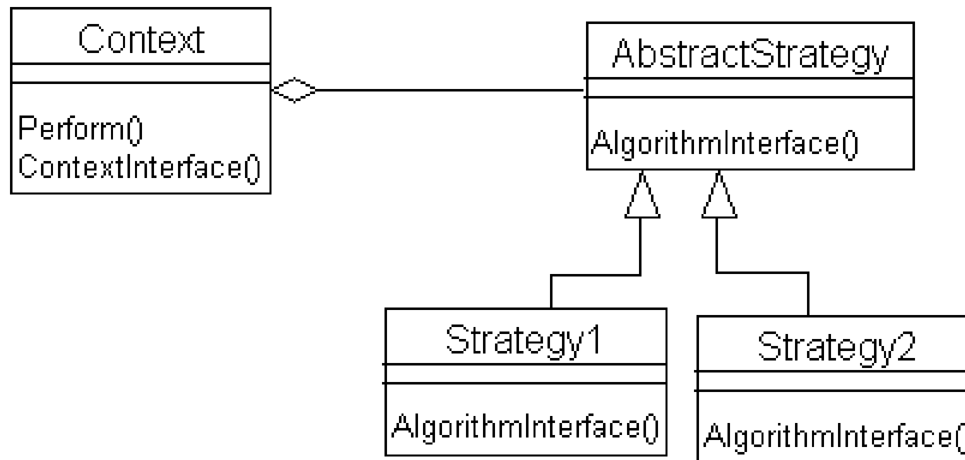
。在实际应用中，工厂方法用得比较多一些，而且是和动态类装入器组合在一起应用

GoF Factory Method模式



- ACE_SOCK_Connector类是一个工厂(factory),用来主动建立一个新的通信端
- ACE_SOCK_Acceptor类是一个工厂(factory),用来被动建立一个新的通信端

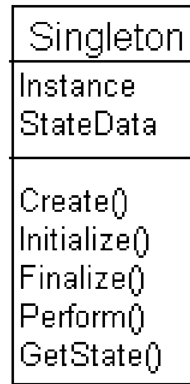
GoF Strategy模式



Strategy使不同算法各自封装,用户端可随意挑选所需要的算法.

ACE_Message_Queue类的构造器及其open()和notification_strategy() 方法，可被用于为ACE_Message_Queue设置通知策略。通知策略派生自ACE_Notification_Strategy

GoF Singleton模式



Singleton保证一个类只有一个实例,并提供一个访问它的全局访问点

1、ACE_Process_Manager

。作为Singleton 使用，要用到instance(), close_singleton()方法

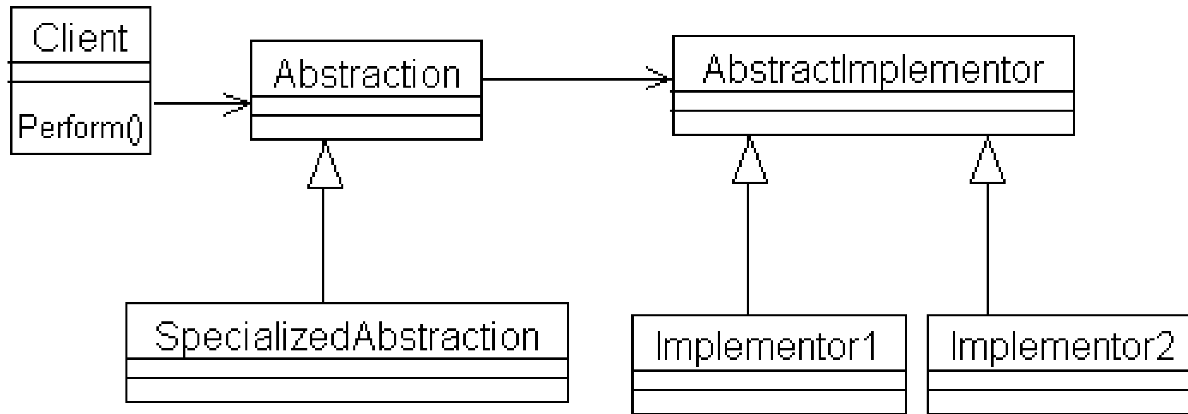
。实例化一个或多个实例，支持“一个进程中的多组进程”

此外,ACE_Process_Manager还应用了Façade模式

2、ACE_Thread_Manager的使用也可以采用上述的两种方法

3、ACE_Reactor 通过Singleton，结合Double-Checked Locking Optimization 模式来创建和管理

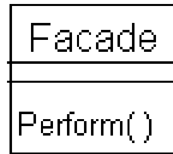
GoF Bridge模式



Bridge将抽象和行为划分开来,各自独立,但能动态的结合.

ACE_Service_Type类使用了Bridge模式来让服务类型中类型特有的数据和行为能够不影响类而进行演化

GoF Facade模式



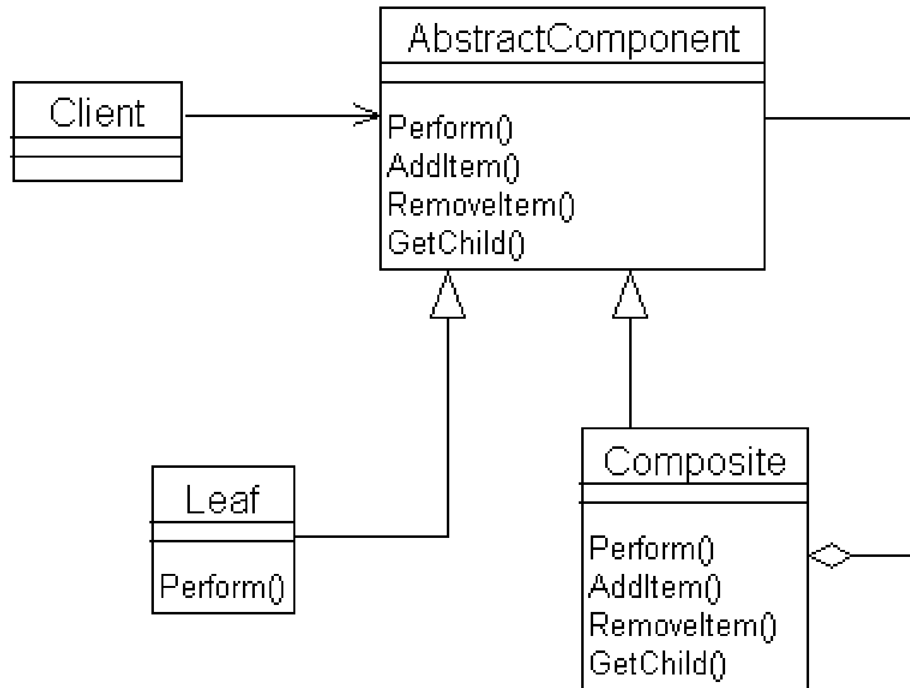
Facade为子系统中的一组接口提供一个一致的界面

。ACE框架可被用于开发较高级的应用组件，而应用组件的接口又为框架的内部结构提供了Façade

。ACE_Reactor实现了Façade模式,定义访问各种ACE_Reactor框架特性的接口

。ACE_Service_Config实现了Façade模式，用来集成ACE Service Configurator框架中的其他类,并对管理应用中的各个服务所必需的活动进行协调。

GoF Composite模式



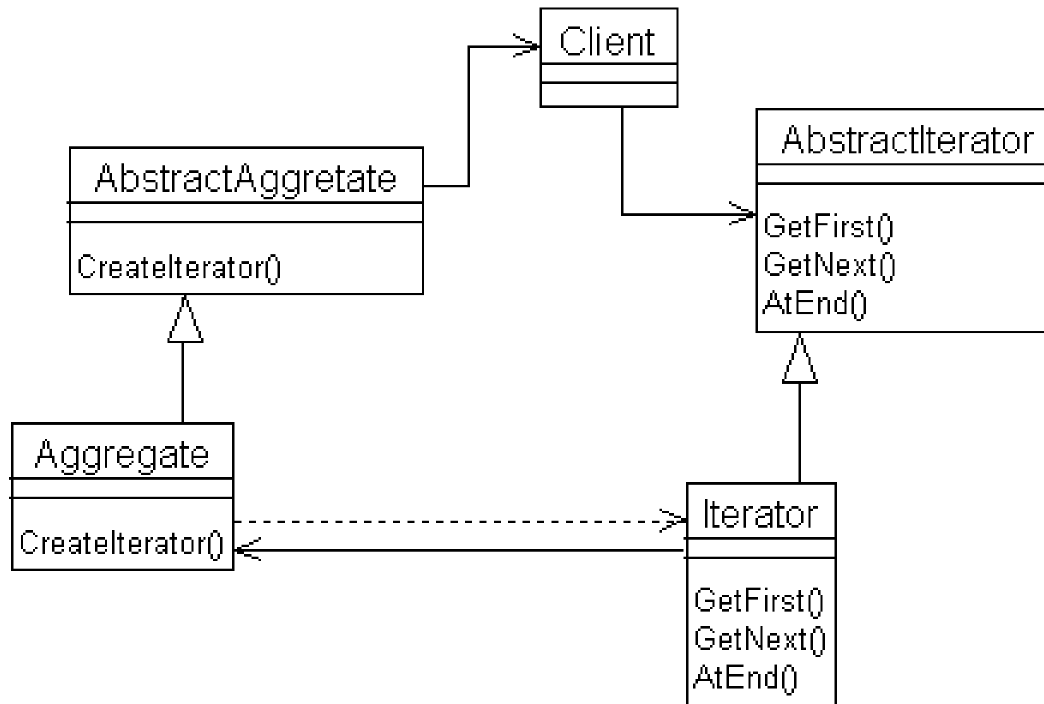
Composite提供用于访问自己组合体内的组件方法: 增加, 删除, 遍历

ACE_Message_Block实现了Composite模式

。如果多个ACE_Message_Block连接在一起(通过Composite模式), 形成单链表, 则形成了复合消息结构.

。将多条消息连接起来, 形成双链表, 则构成ACE_Message_Queue类

GoF Iterator模式

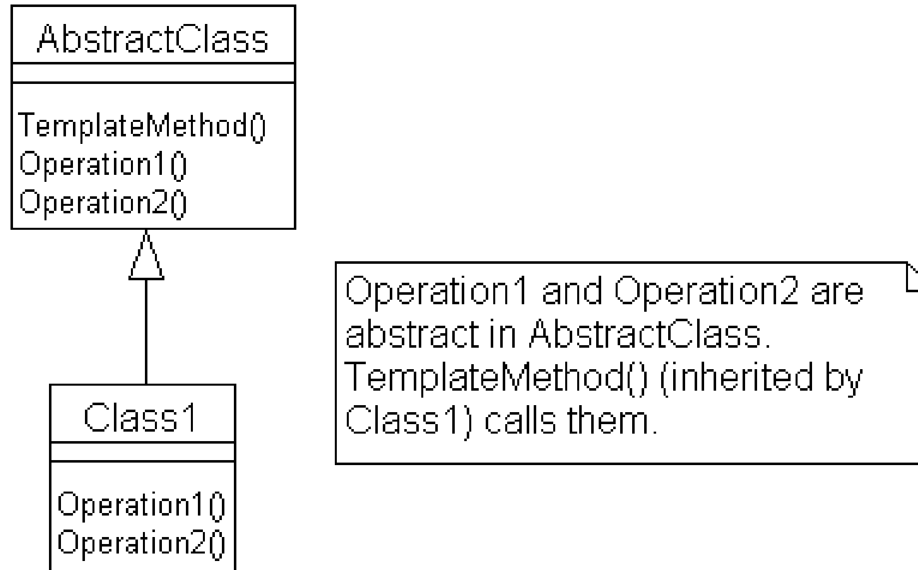


使用迭代(Iterator)进行对象遍历

。ACE_Handle_Set_Iterator基于Iterator模式。它提供了连续访问聚合对象(aggregate object)中元素的方法

。这个类可以高效地搜索ACE_Handle_Set中的句柄,并运用operator()方法，一次只返回一个活动句柄

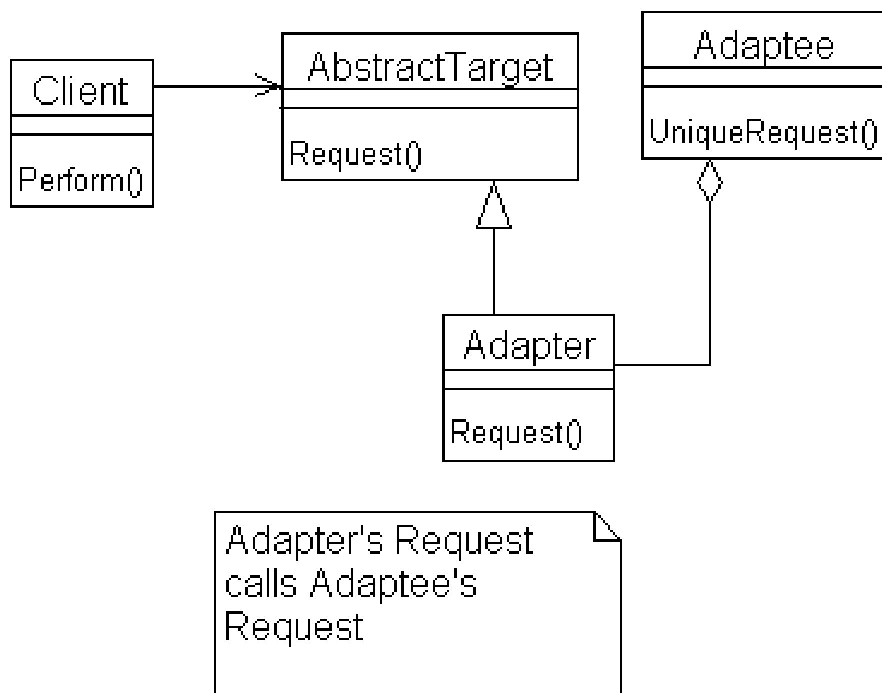
GoF Template Method模式



Template定义一个操作中算法的骨架,将一些步骤的执行延迟到其子类中

ACE在很多地方使用了Template Method 它定义了“一个操作中的一个算法”的骨架

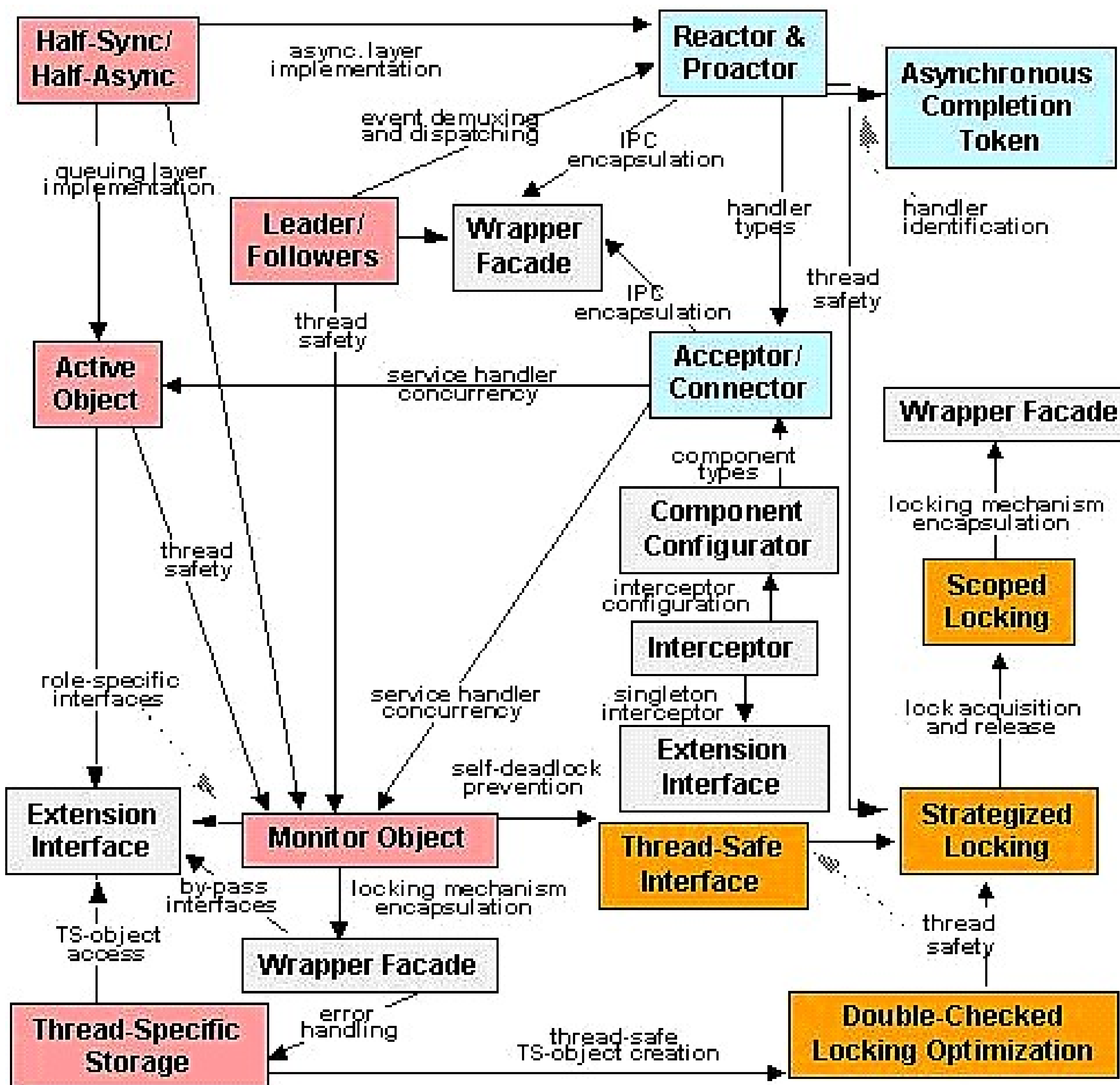
GoF Adapter模式



将两个不兼容的类合在一起使用, 属于结构型模式, 需要有Adaptee(被适配者)和Adapter(适配器)两个身份.

。如ACE的日志服务程序中，就使用Adapter将记录重定向到UNIX的Syslogd,或Windows NT/2000的事件记录。
。ACE_Reactor类使用了Bridge模式使其实现与类接口分离开来

ACE模式和框架



http://www.cs.wustl.edu/~schmidt/patterns.html

Stream(流):分层服务的集成

ACE Stream组件简化了那些分层的(layered)或层次的(hierarchic)软件的开发。

流类属含有自适配服务执行体（ASX）构架，它集成了较低级的OO包装组件（像IPC SAP）和较高级的类属（像Reactor和Service Configurator）。

ASX构架合并了来自若干模块化的通信构架的概念，其中包括系统V STREAMS、x-kernel和来自面向对象操作系统Choices的Conduit构架。

ASX构架为通信软件的开发者提供下面两种好处：

1.它嵌入、封装，并实现了通常用于开发通信软件的一些关键设计模式。

2.它严格地区分了关键的开发事务

流类属的主要组件:

* ACE_Stream
* ACE_Module
* ACE_Task

服务配置器(Service Configurator)

ACE提供Service Configurator类属来在一组类和继承层次中封装SunOS的显式链接机制。

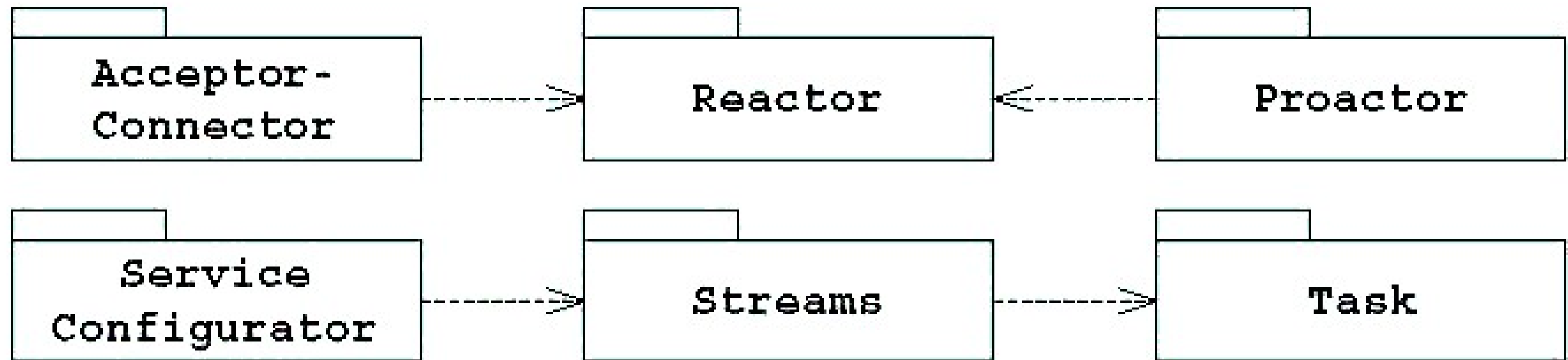
Service Configurator中的主要配置单元是服务(service)

包含的组件:
--ACE_Service_Object
--ACE_Event_Handler
--ACE_Shared_Object
--ACE_Service_Repository
--ACE_Service_Config (svc.conf)

所有的应用服务都派生自ACE_Service_Object继承层次

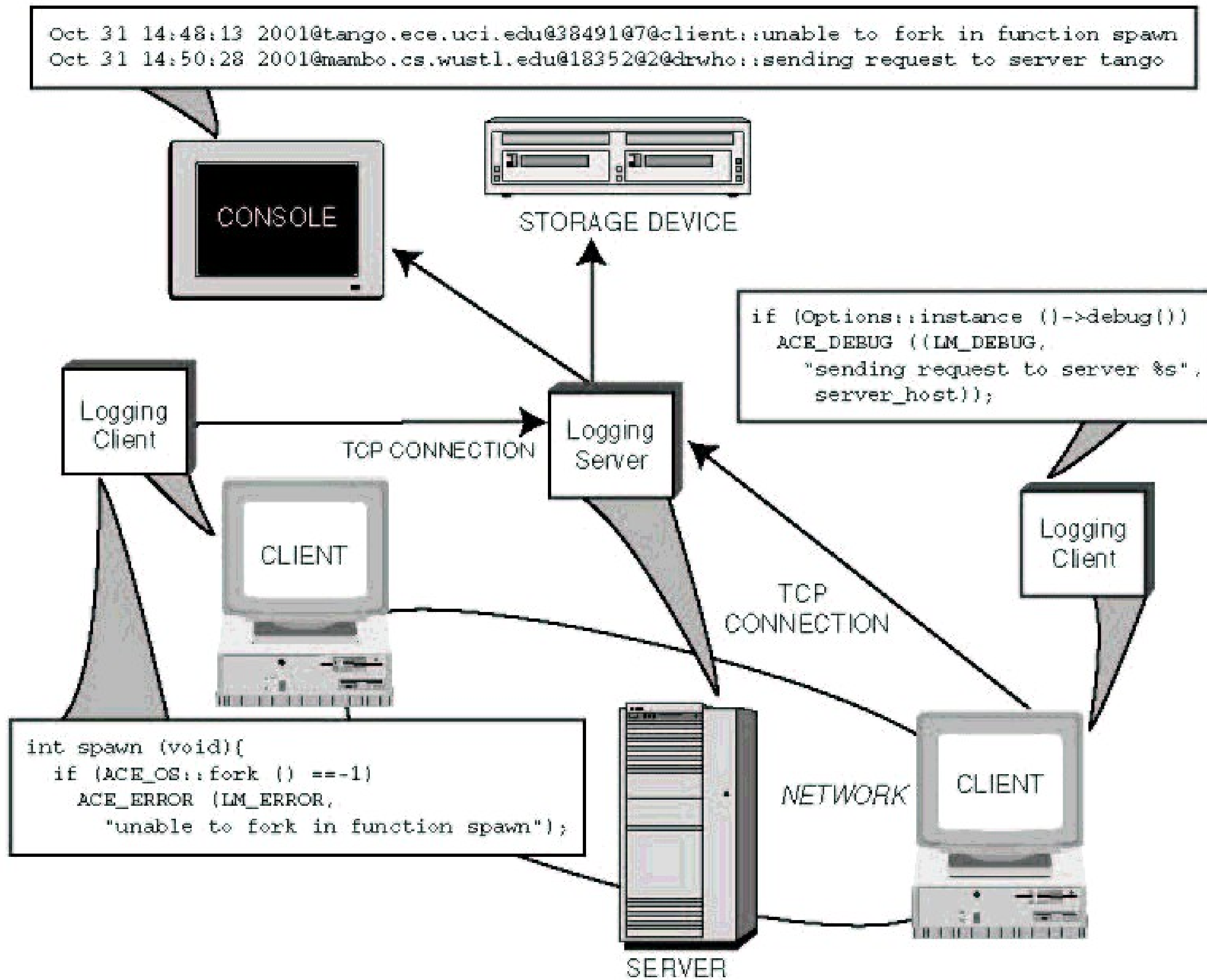
ACE_Service_Object继承层次由ACE_Event_Handler和ACE_Shared_Object抽象基类组成。

ACE框架小结



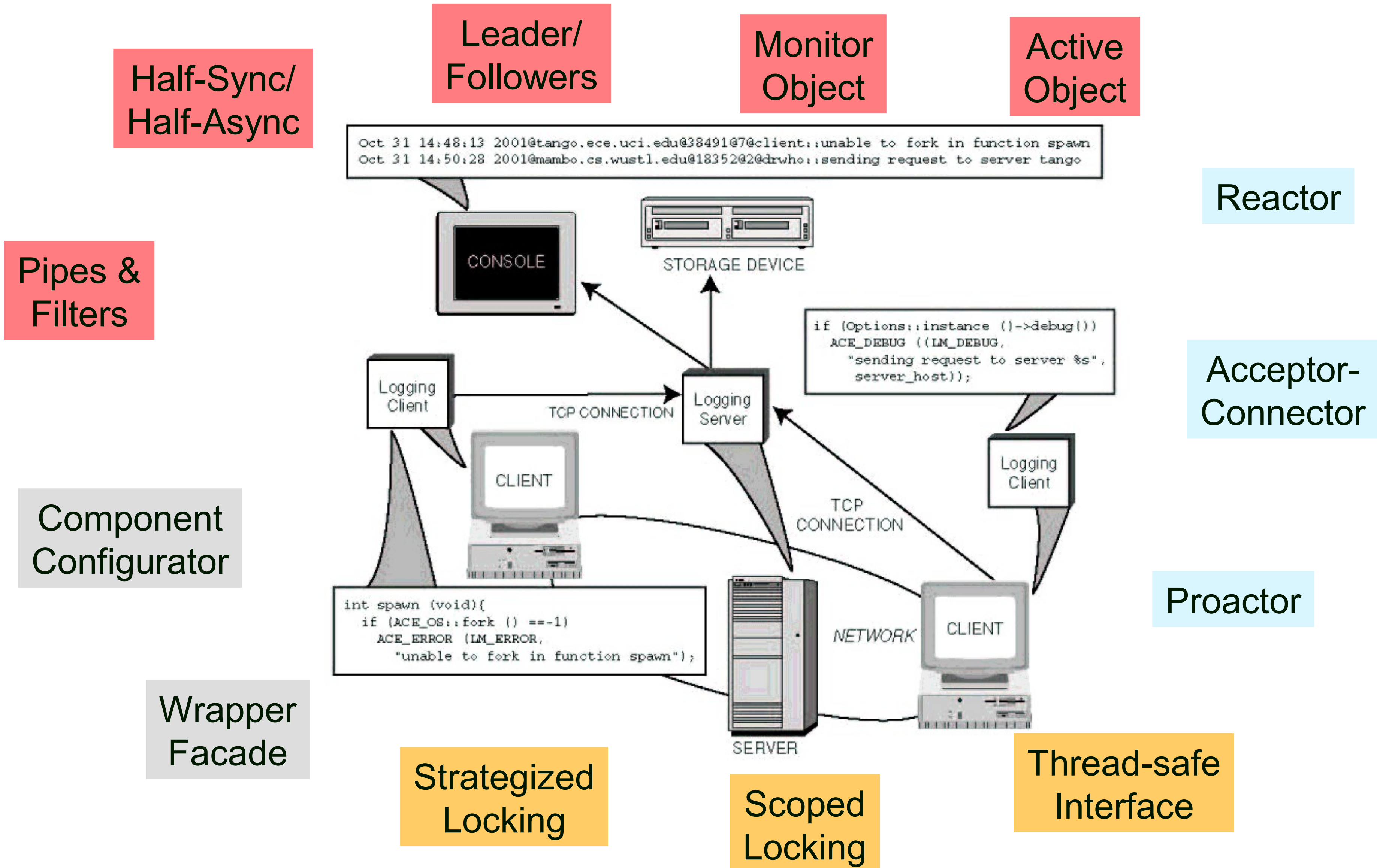
ACE Framework	**Inversion of Control**
Reactor & Proactor	Calls back to application-supplied event handlers to perform processing when events occur synchronously & asynchronously
Service Configurator	Calls back to application-supplied service objects to initialize, suspend, resume, & finalize them
Task	Calls back to an application-supplied hook method to perform processing in one or more threads of control
Acceptor-Connector	Calls back to service handlers to initialize them after they are connected
Streams	Calls back to initialize & finalize tasks when they are pushed & popped from a stream

分布式日志工具



ACE提供了一个分布式日志工具，以简化调试和运行时跟踪。该工具目前被用于一个商业在线事务处理系统中,为高速网络环境中的群集工作站和多处理器数据库服务器提供日志服务。

分布式日志中的模式



ACE Logging

基础**Logging and Tracing**
三个宏:ACE_DEBUG, ACE_ERROR, ACE_TRACE

ACE_DEBUG ((severity, formatting-args));
ACE_ERROR ((severity, formatting-args));

如：ACE_DEBUG ((debug_info << "Hi ACE" << endl));

严重级别	含义
LM_TRACE	指示函数调用次序的消息
LM_DEBUG	调试消息
LM_INFO	消息含有通常只在程序调试时使用的信息
LM_NOTICE	不是出错的情况，而是可能需要特别处理的情况
LM_WARNING	警告消息
LM_ERROR	错误消息
LM_CRITICAL	紧急情况，比如硬件设备出错
LM_ALERT	应该立即纠正的情况，比如系统数据库损坏
LM_EMERGENCY	万分紧急的情况，通常会广播给所有用户

ACE Logging examples

```
#include "ace/Log_Msg.h"

void foo (void);

int ACE_TMAIN (int, ACE_TCHAR *[])
{
  ACE_TRACE(ACE_TEXT ("main"));

  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%I Hi Everyone\n")));
  foo();
  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%I Goodnight\n")));

  return 0;
}

void foo (void)
{
  ACE_TRACE (ACE_TEXT ("foo"));

  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%I Huihoo Power!\n")));
}
```

Simple1.cpp

重定向 Logging输出

- The system logger (UNIX syslog or NT Event Log)
- A programmer-specified output stream, such as a file

```
int ACE_TMAIN (int, ACE_TCHAR *argv[])
{
  ACE_LOG_MSG->open
    (argv[0], ACE_Log_Msg::SYSLOG, ACE_TEXT ("ACE-syslogTest"));

  ACE_TRACE (ACE_TEXT ("main"));

  ACE_DEBUG ((LM_DEBUG, ACE_TEXT ("%I Hi Everyone\n")));

  ACE_DEBUG ((LM_INFO, ACE_TEXT ("%IGoodnight\n")));

  return 0;
}
```

syslog.cpp

打开windows的事件察看器>应用程序日志 ,可看到已插入几条日志

使用 Callbacks

```
// Callback.h

#include "ace/streams.h"
#include "ace/Log_Msg.h"
#include "ace/Log_Msg_Callback.h"
#include "ace/Log_Record.h"

class Callback : public ACE_Log_Msg_Callback
{
public:
  void log (ACE_Log_Record &log_record) {
    printf("%s\n",log_record.msg_data());
    log_record.print (ACE_TEXT (""), ACE_Log_Msg::VERBOSE, cerr);
  }
};

  Callback call;
  ACE_LOG_MSG->msg_callback((ACE_Log_Msg_Callback*)&call);
```

Use_Callback.cpp

Logging Client and Server Daemons

ACE netsvcs 日志框架有一个 client/server design.

server.conf
dynamic Logger Service_Object * ACE:_make_ACE_Logging_Strategy() "-s foobar -f STDERR|OSTREAM|VERBOSE"

dynamic Server_Logging_Service Service_Object * netsvcs:_make_ACE_Server_Logging_Acceptor () active "-p 20009"

运行：
$ACE_ROOT/netsvcs/servers/main -f server.conf (Linux,UNIX) or
%ACE_ROOT%\netsvcs\servers\main –f server.conf (windows)

client.conf
dynamic Client_Logging_Service Service_Object * netsvcs:_make_ACE_Client_Logging_Acceptor () active "-p 20009 -h localhost"

运行：
$ACE_ROOT/netsvcs/servers/main -f client.conf (Linux,UNIX) or
%ACE_ROOT%\netsvcs\servers\main –f client.conf (windows)

收集运行时信息

有两种方式收集运行时信息

- 接受命令行arguments and options.
- 读取configuration files.

- ACE_Get_Opt: to access command line arguments and options
- ACE_Configuration: to manipulate configuration information on all platforms using the ACE_Configuration_Heap class and, for the Windows registry, the ACE_Configuration_Win32Registry class

ACE_Get_Opt

```
static const ACE_TCHAR options[] = ACE_TEXT (":f:");
ACE_Get_Opt cmd_opts (argc, argv, options);
if (cmd_opts.long_option
    (ACE_TEXT ("config"), 'f', ACE_Get_Opt::ARG_REQUIRED) == -1)
  return -1;
int option;
ACE_TCHAR config_file[MAXPATHLEN];
ACE_OS_String::strcpy (config_file, ACE_TEXT ("HAStatus.conf"));
while ((option = cmd_opts ()) != EOF)
  switch (option) {
  case 'f':
    ACE_OS_String::strncpy (config_file,
                            cmd_opts.opt_arg (),
                            MAXPATHLEN);
    break;
  case ':':
    ACE_ERROR_RETURN
      ((LM_ERROR, ACE_TEXT ("-%c requires an argument\n"),
        cmd_opts.opt_opt ()), -1);
  default:
    ACE_ERROR_RETURN
      ((LM_ERROR, ACE_TEXT ("Parse error.\n")), -1);
  }
```

ACE_Configuration_Heap

```
ACE_Configuration_Heap config;
if (config.open () == -1)
  ACE_ERROR_RETURN
    ((LM_ERROR, ACE_TEXT ("%p\n"), ACE_TEXT ("config")), -1);
ACE_Registry_ImpExp config_importer (config);
if (config_importer.import_config (config_file) == -1)
  ACE_ERROR_RETURN
    ((LM_ERROR, ACE_TEXT ("%p\n"), config_file), -1);

ACE_Configuration_Section_Key status_section;
if (config.open_section (config.root_section (),
                         ACE_TEXT ("HAStatus"),
                         0,
                         status_section) == -1)
  ACE_ERROR_RETURN ((LM_ERROR, ACE_TEXT ("%p\n"),
                     ACE_TEXT ("Can't open HAStatus section")),
                    -1);

u_int status_port;
if (config.get_integer_value (status_section,
                              ACE_TEXT ("ListenPort"),
                              status_port) == -1)
  ACE_ERROR_RETURN
    ((LM_ERROR,
      ACE_TEXT ("HAStatus ListenPort does not exist\n")),
     -1);
this->listen_addr_.set (ACE_static_cast (u_short, status_port));
```

ACE容器

ACE 支持以下两类容器：
--template-based type-safe containers
Template-based containers use the C++ templates facility, which allows you to create a "type-specific" container at compile time. For example, if you wanted to store information about all the people in a household, you could create a People list that would allow insertion of only People objects into the list.

--object-based containers
Object-based containers support insertion and deletion of a class of object types. If you have programmed with Java or Smalltalk, you will recognize these containers as supporting insertion of the generic object type. ACE has a few containers of this type, built for specific uses, such as the ACE_Message_ Queue class. We will not be discussing these in this chapter but instead will defer the discussion until their specific use comes up.

ACE提供了一套容器类:
- 单链表和双链表
- 集和多重集
- 栈和队列
- 动态数组
- 字符串处理类

Sequence 容器类

- **Doubly Linked List**

ACE_DLList

DLList.exe

- **Stacks**

ACE_Bounded_Stack，ACE_Fixed_Stack

Stacks.exe

- **Queues**

ACE_Unbounded_Queue

Queues.exe

- **Arrays**

ACE_Array

Array.exe

- **Sets**

runBoundedSet() , runUnboundedSet()

Sets.exe

ACE List

```
#include "ace/Containers.h"

int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
    // 创建一个双向列表
    ACE_DLList<int> intList;
    for (int i = 0; i < 10;i++)
    {
        int* p;
        ACE_NEW_RETURN(p,int(i),-1);
        ACE_DEBUG((LM_DEBUG,ACE_TEXT("%x\n"),p));
        intList.insert_head(p);
    }
    ACE_DEBUG((LM_DEBUG,ACE_TEXT("\n")));

    // 创建一个迭代器
    ACE_DLList_Iterator<int>intIter(intList);
    while (!intIter.done())
    {
        ACE_DEBUG((LM_DEBUG,ACE_TEXT("%x\n"),*(intIter.next())));
        intIter++;
    }
....
```

Associative 容器类

。**Map Manager**

ACE_Map_Manager

Map_Manager.exe

。**Hash Maps**

ACE_Hash_Map_Manager

Hash_Map_Hash.exe

。**Self-Adjusting Binary Tree**

ACE_RB_Tree

RB_Tree.exe

注：ACE_Hash_Map_Manager 定义了一个集抽象体,将”键”和”值”有效的关联起来,它基于散列技术(hashing)执行高效搜索,而std::map不支持这一点.所以没有使用标准的std::map

ACE成功应用

-- 基于ACE构建的TAO ,实时CORBA的领导者

-- ACE已在波音被用于构建实时航空控制系统

http://www.cs.wustl.edu/~schmidt/TAO-boeing.html

-- 爱立信、摩托罗拉和朗讯将ACE用于电信系统

-- 华为的网管软件是用ACE,SNACC,SNMP++做的。

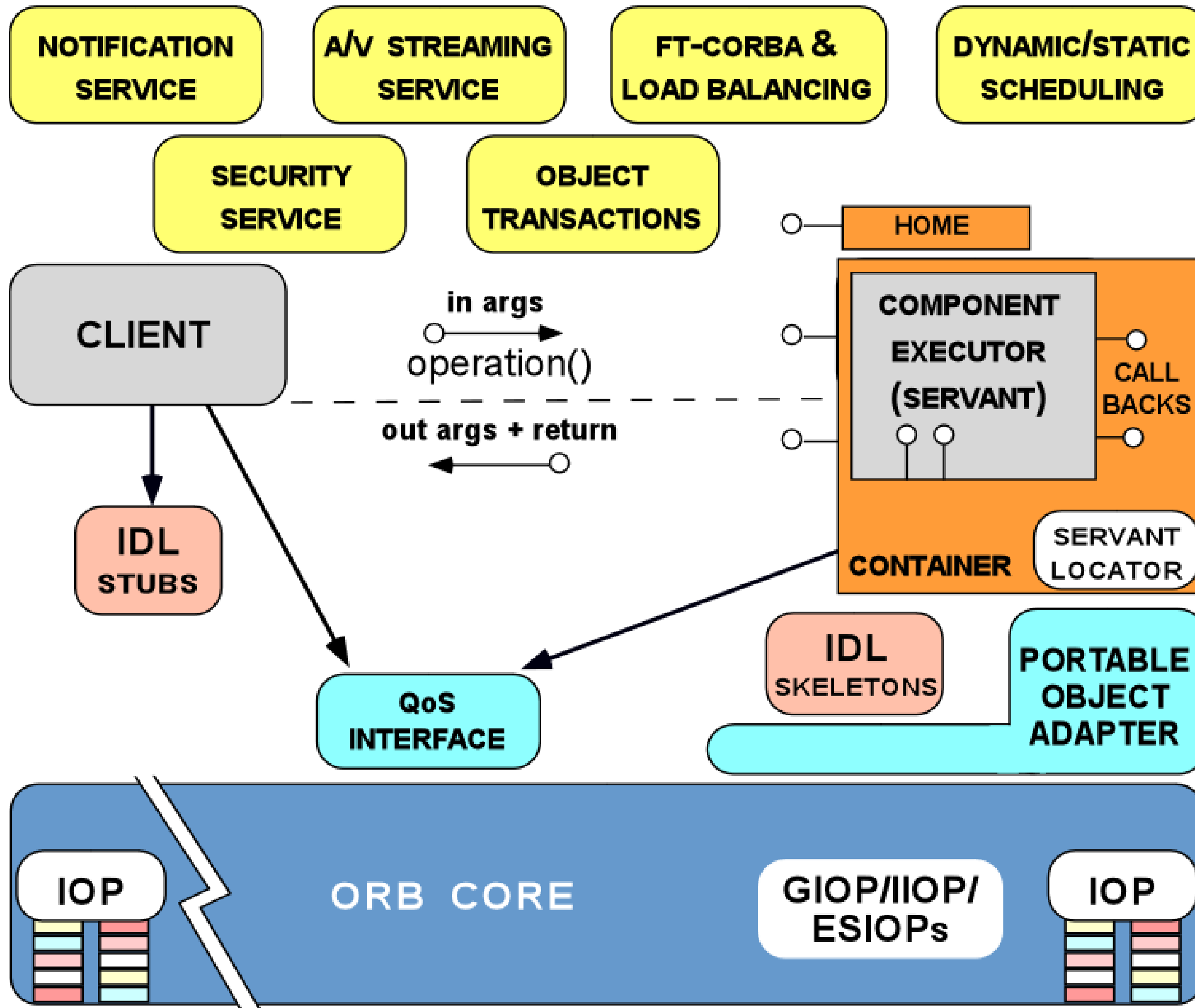
-- 西门子和柯达将ACE用于医学成像系统

-- 用于热轨碾扎的工业环境 http://**www.siroll.de**

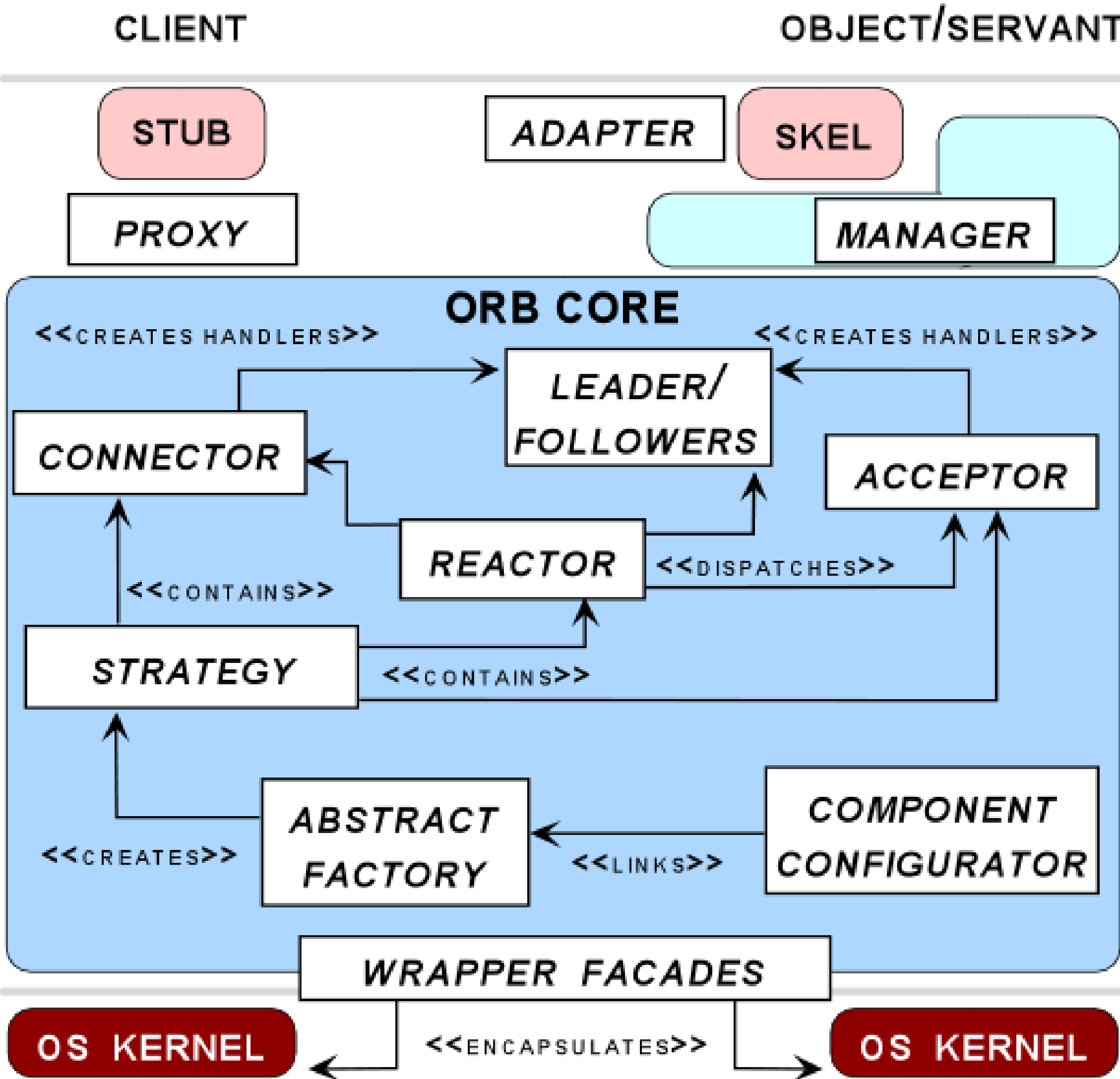
-- 用于大规模路由器 http://www.arl.wustl.edu/

-- 基于ACE构建的高性能Web服务器 JAWS

TAO（实时CORBA）

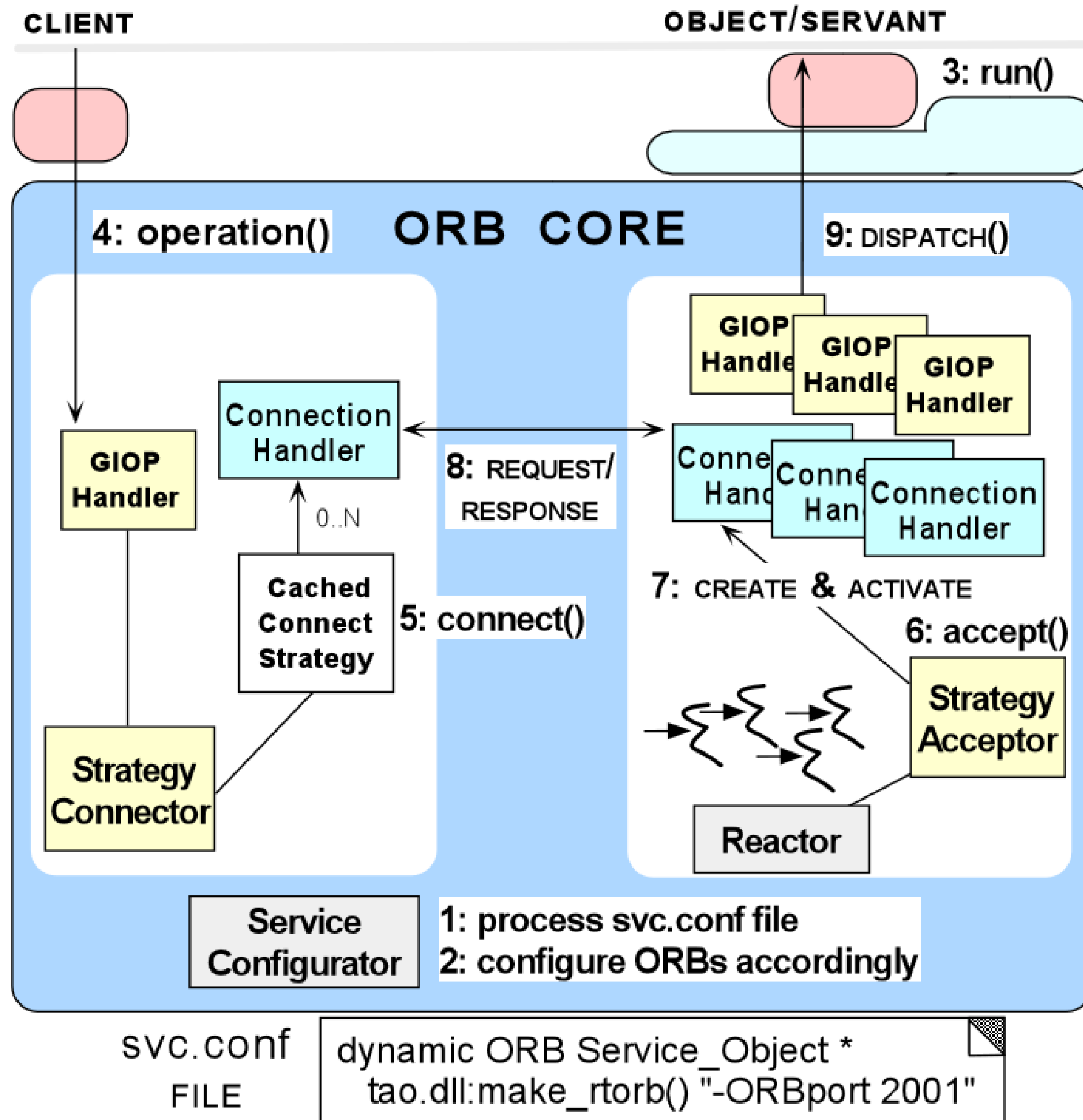


TAO中使用的核心模式



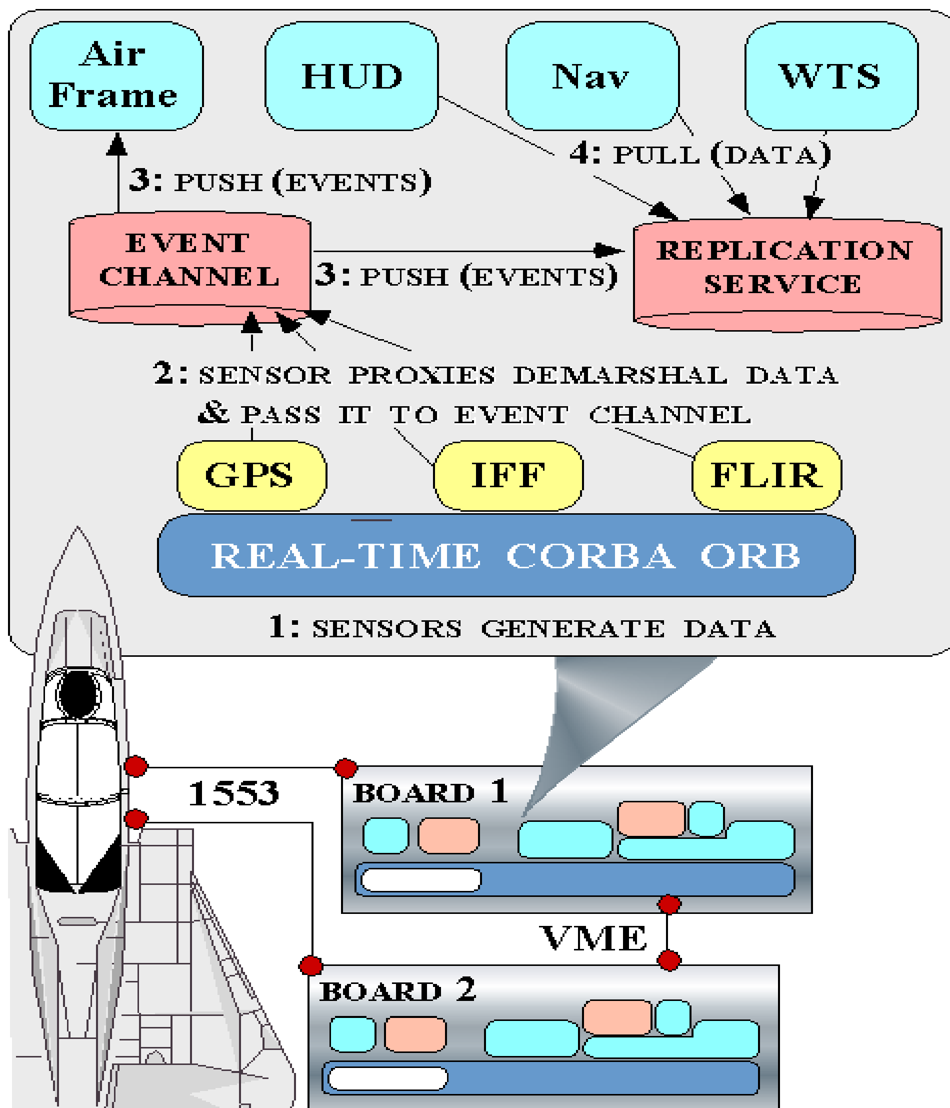
- ***Wrapper facades*** enhance portability
- ***Proxies & adapters*** simplify client & server applications, respectively
- ***Component Configurator*** dynamically configures ***Factories***
- ***Factories*** produce ***Strategies***
- ***Strategies*** implement interchangeable policies
- Concurrency strategies use ***Reactor*** & ***Leader/Followers***
- ***Acceptor-Connector*** decouples connection management from request processing
- ***Managers*** optimize request demultiplexing

TAO中使用的ACE框架



- ***Reactor*** drives the ORB event loop
  - Implements the ***Reactor*** & ***Leader/Followers*** patterns
- ***Acceptor-Connector*** decouples passive/active connection roles from GIOP request processing
  - Implements the ***Acceptor-Connector*** & ***Strategy*** patterns
- ***Service Configurator*** dynamically configures ORB strategies
  - Implements the ***Component Configurator*** & ***Abstract Factory*** patterns

波音—实时航空控制系统



目的

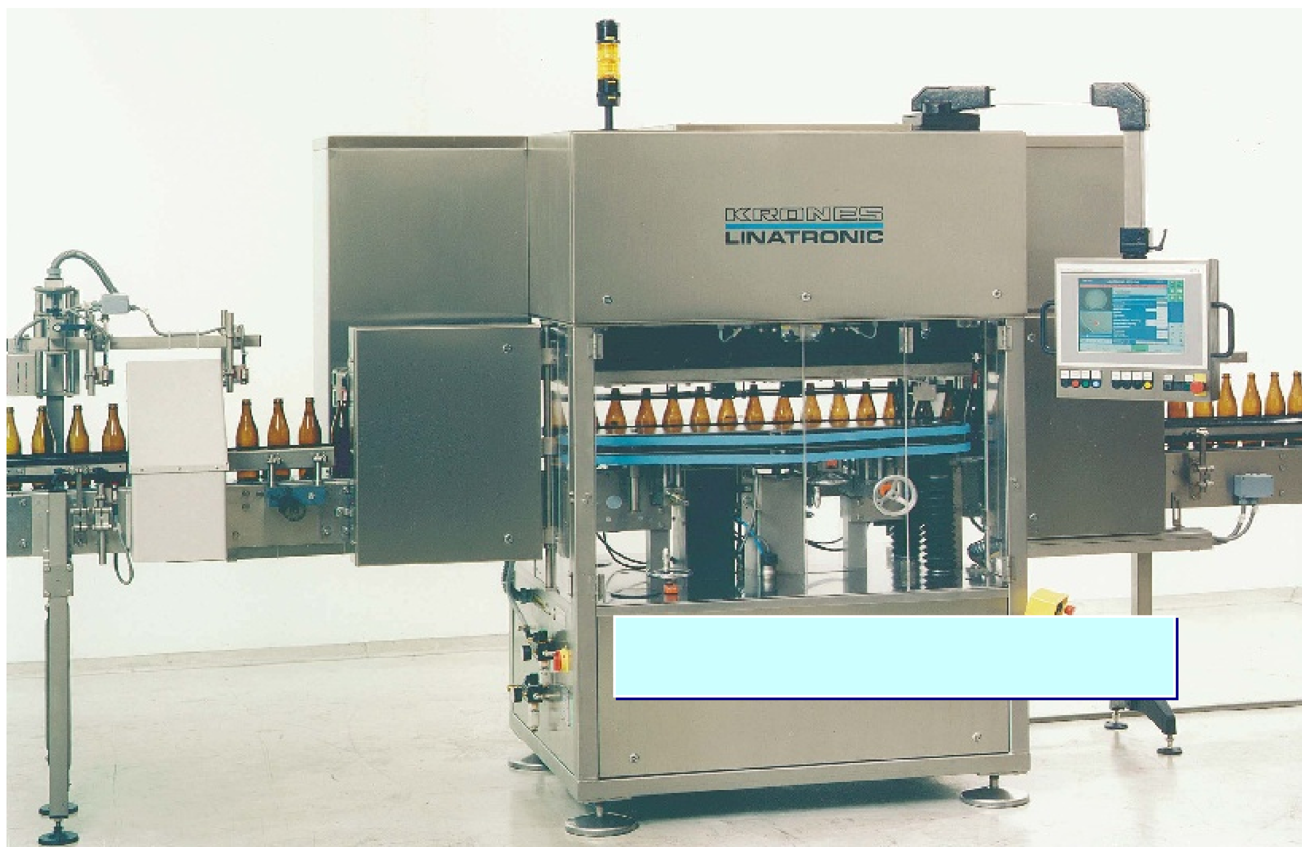
- 应用COTS & open systems 于实时航空控制系统

Key Results

- First use of RT CORBA in mission computing
- Drove Real-time CORBA standardization

www.boeing.com

实时图像处理



目的

- 实时检查瓶子的缺陷

Key Software Solution Characteristics

- Affordable, flexible, & COTS
- Embedded Linux (Lem)
- Compact PCI bus + Celeron processors
- Remote booted by DHCP/TFTP
- Real-time CORBA (ACE+TAO)

www.krones.com

热轨碾扎

目的

- 实时控制熔化的钢板在热轨碾扎机上的移动

Key Software Solution Characteristics

- Affordable, flexible, & COTS
- Product-line architecture
- Design guided by patterns & frameworks
- Windows NT/2000
- Real-time CORBA (ACE+TAO)

www.siroll.de

内容回顾

- ACE介绍
  - ACE的获得和安装
  - ACE综述
  - ACE架构层次
  - ACE OS适配层
  - ACE OO包装
- GoF模式：

Strategy, Singleton, Bridge, Facade, Composite 的介绍

内容回顾

- ACE模式

Reactor,Proactor,Component Configurator,

Active Object,Half-Sync/Half-Async,Acceptor-onnector,

Pipes and Filters

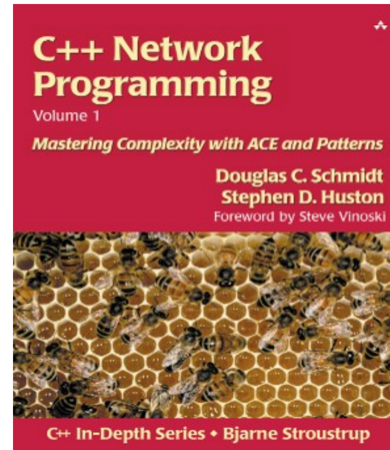
- 使用ACE的日志服务
- 收集运行时信息
- ACE容器
- ACE成功应用

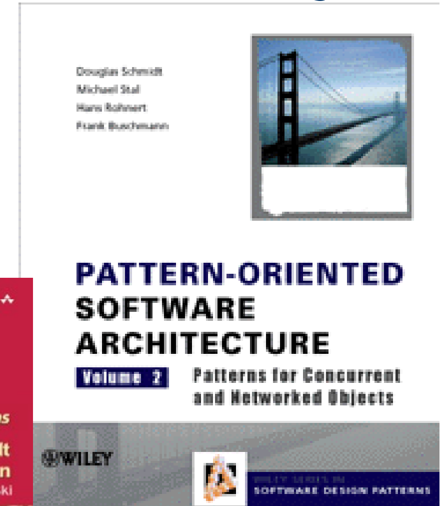
总结

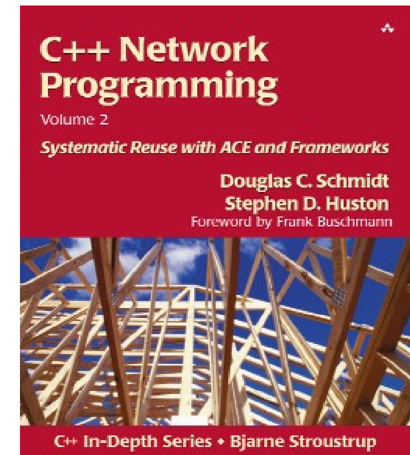
- ACE包含操作系统(OS)适配层, C++包装层,构架和模式层三个层次的内容
- ACE可以和JVM,CLR放在一个层次上加以考虑
- ACE适合高性能和实时通信服务和应用的开发
- ACE增强应用系统的可移植性
- 通过ACE的关键模式更好的提高软件质量
- 更高的效率和可预测性,支持广泛的应用服务质量（QoS）需求
- ACE和TAO协同工作, 提供全面的中间件解决方案

参考资料

- Patterns & frameworks for concurrent & networked objects
  - www.posa.uci.edu
- ACE & TAO open-source middleware
  - www.cs.wustl.edu/~schmidt/ACE.html
  - www.cs.wustl.edu/~schmidt/TAO.html







- ACE research papers
  - www.cs.wustl.edu/~schmidt/ACE-papers.html
- Extended ACE & TAO tutorials
  - UCLA extension, January 21-23, 2004
  - www.cs.wustl.edu/~schmidt/UCLA.html
- ACE books
  - www.cs.wustl.edu/~schmidt/ACE/

结束

谢谢大家！

ihuihoo@gmail.com
http://www.huihoo.com