第2章 包装外观（Wrapper Facade）：用于在类中封装函数的结构型模式

Douglas C. Schmidt

2.1 介绍

本论文描述包装外观模式。该模式的意图是通过面向对象（OO）类接口来封装低级函数和数据结构。常见的包装外观模式的例子是像MFC、ACE和AWT这样的类库，它们封装本地的OS C API，比如socket、pthreads或GUI函数。

直接对本地OS C API进行编程会使网络应用繁琐、不健壮、不可移植，且难以维护，因为应用开发者需要了解许多低级、易错的细节。本论文阐释包装外观模式怎样使这些类型的应用变得更为简洁、健壮、可移植和可维护。

本论文被组织如下：2.2详细描述使用西门子格式[1]的包装外观模式，2.3给出结束语。

2.2 包装外观模式

2.2.1 意图

在更为简洁、健壮、可移植和可维护的较高级面向对象类接口中封装低级函数和数据结构。

2.2.2 例子

为阐释包装外观模式，考虑图2-1中所示的分布式日志服务的服务器。客户应用使用日志服务来记录关于它们在分布式环境中的执行状态的信息。这些状态信息通常包括错误通知、调试跟踪和性能诊断。日志记录被发送到中央日志服务器，由它将记录写到各种输出设备，比如网络管理控制台、打印机或数据库。

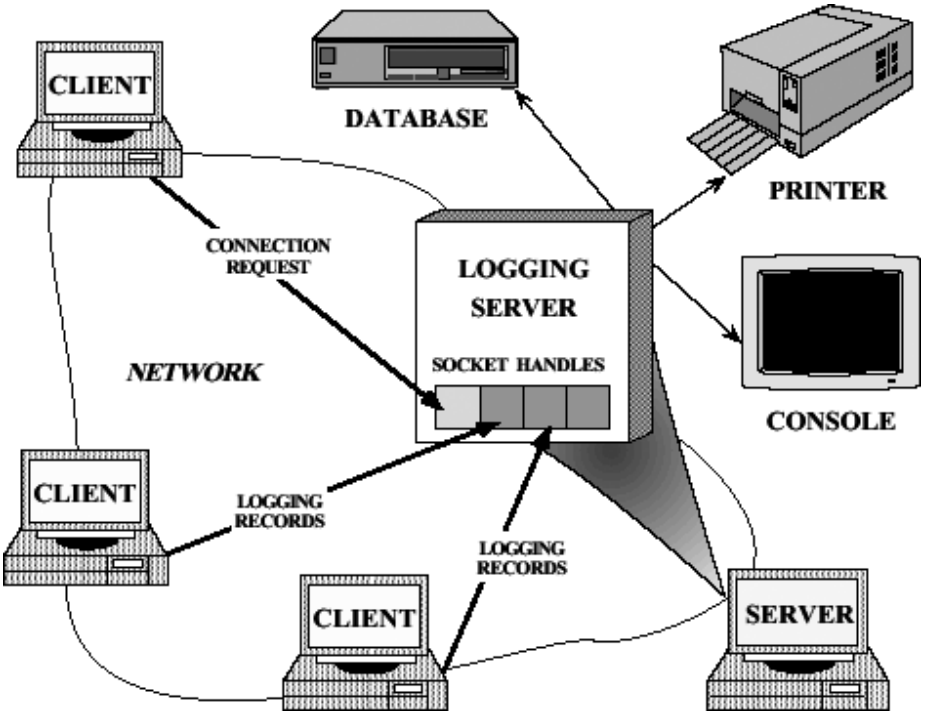

图2-1 分布式日志服务

图2-1所示的日志服务器处理客户发送的连接请求和日志记录。日志记录和连接请求可以并发地在多个socket *句柄*上到达。各个句柄标识在OS中管理的网络通信资源。

客户使用像TCP[2]这样的面向连接协议与日志服务器通信。因而，当客户想要记录日志数据时，它必须首先向日志服务器发送连接请求。服务器使用*句柄工厂*（handle factory）来接受连接请求，句柄工厂在客户知道的网络地址上进行侦听。当连接请求到达时，OS句柄工厂接受客户的连接，并创建表示该客户的连接端点的socket句柄。该句柄被返回给日志服务器，后者在这个句柄和其他句柄上等待客户日志请求到达。一旦客户被连接，它们可以发送日志记录给服务器。服务器通过已连接的socket句柄来接收这些记录，处理记录，并将它们写到它们的输出设备。

开发并发处理多个客户的日志服务器的常见方法是使用低级C语言函数和数据结构来完成线程、同步及网络通信等操作。例如，图2-2演示怎样将Solaris线程[3]和socket[4]网络编程API用于开发多线程日志服务器。

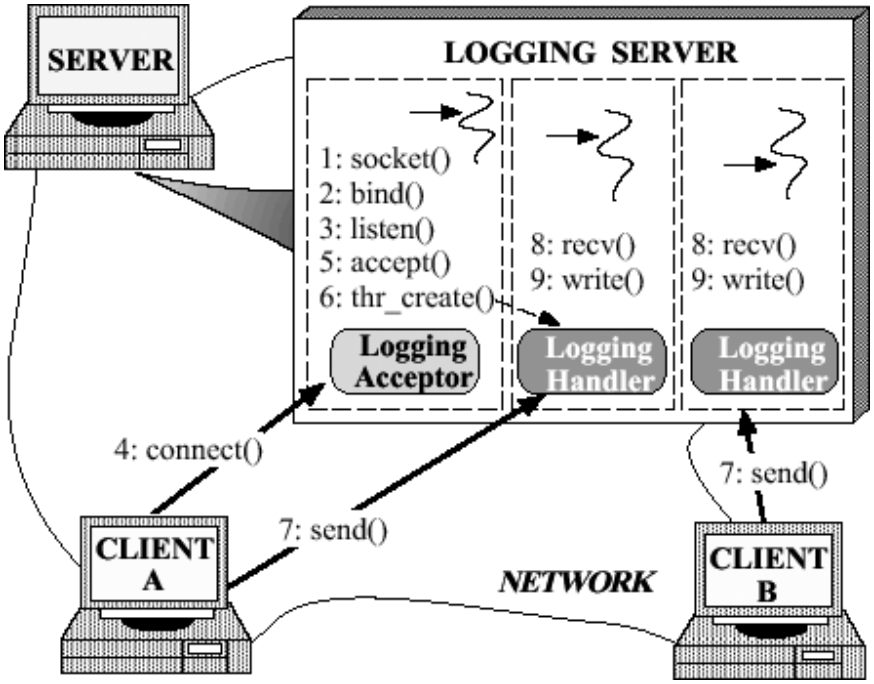

图2-2 多线程日志服务器

在此设计中，日志服务器的句柄工厂在它的主线程中接受客户网络连接。它随即派生一个新线程，在单独的连接中运行logging_handler函数、以处理来自每个客户的日志记录。下面的两个C函数演示怎样使用socket、互斥体和线程的本地Solaris OS API来实现此日志服务器设计。

```
// At file scope.
// Keep track of number of logging requests.
static int request_count;
// Lock to protect request_count.
static mutex_t lock;
// Forward declaration.
static void *logging_handler (void *);
// Port number to listen on for requests.
static const int logging_port = 10000;

// Main driver function for the multi-threaded
// logging server. Some error handling has been
// omitted to save space in the example.
int main (int argc, char *argv[])
{
struct sockaddr_in sock_addr;
```

```
// Handle UNIX/Win32 portability differences.
#if defined (_WINSOCKAPI_)
SOCKET acceptor;
#else
int acceptor;
#endif /* _WINSOCKAPI_ */

// Create a local endpoint of communication.
acceptor = socket (PF_INET, SOCK_STREAM, 0);

// Set up the address to become a server.
memset (reinterpret_cast <void *> (&sock_addr), 0, sizeof sock_addr);
sock_addr.sin_family = AF_INET;
sock_addr.sin_port = htons (logging_port);
sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);

// Associate address with endpoint.
bind (acceptor,
       reinterpret_cast <struct sockaddr *>
       (&sock_addr),
              sizeof sock_addr);

// Make endpoint listen for connections.
listen (acceptor, 5);

// Main server event loop.
for (;;)
{
       thread_t t_id;

       // Handle UNIX/Win32 portability differences.
#if defined (_WINSOCKAPI_)
       SOCKET h;
#else
       int h;
```

```
#endif /* _WINSOCKAPI_ */

       // Block waiting for clients to connect.
       int h = accept (acceptor, 0, 0);

       // Spawn a new thread that runs the
       // <logging_handler> entry point and
       // processes client logging records on
       // socket handle <h>.
       thr_create (0, 0,
                           logging_handler,
                     reinterpret_cast <void *> (h),
                     THR_DETACHED,
                     &t_id);
}

/* NOTREACHED */
return 0;
}
```

logging_handler函数运行在单独的线程控制中，也就是，每个客户一个线程。它在各个连接上接收并处理日志记录，如下所示：

```
// Entry point that processes logging records for
// one client connection.
void *logging_handler (void *arg)
{
// Handle UNIX/Win32 portability differences.
#if defined (_WINSOCKAPI_)
SOCKET h = reinterpret_cast <SOCKET> (arg);
#else
int h = reinterpret_cast <int> (arg);
#endif /* _WINSOCKAPI_ */
for (;;)
{
       UINT_32 len; // Ensure a 32-bit quantity.
```

```
char log_record[LOG_RECORD_MAX];

// The first <recv> reads the length
// (stored as a 32-bit integer) of
// adjacent logging record. This code
// does not handle "short-<recv>s".
ssize_t n = recv
                            (h,
                            reinterpret_cast <char *> (&len),
                            sizeof len, 0);

// Bail out if we don't get the expected len.
if (n <= sizeof len) break;
len = ntohl (len); // Convert byte-ordering.
if (len > LOG_RECORD_MAX) break;

// The second <recv> then reads <len>
// bytes to obtain the actual record.
// This code handles "short-<recv>s".
for (size_t nread = 0;
       nread < len;
       nread += n)
{
       n = recv (h, log_record + nread, len - nread, 0);
       // Bail out if an error occurs.
       if (n <= 0) return 0;
}

mutex_lock (&lock);

// Execute following two statements in a
// critical section to avoid race conditions
// and scrambled output, respectively.
++request_count; // Count # of requests received.

if (write (1, log_record, len) == -1)
```

```
            // Unexpected error occurred, bail out.
            break;

        mutex_unlock (&lock);
}

close (h);
return 0;
}
```

注意全部的线程、同步及网络代码是怎样使用Solaris操作系统所提供的低级C函数和数据结构来编写的。

2.2.3 上下文

访问由低级函数和数据结构所提供服务的应用。

2.2.4 问题

网络应用常常使用2.2.2中所演示的低级函数和数据结构来编写。尽管这是一种惯用方法，由于不能解决以下问题，它会给应用开发者造成许多问题：

繁琐、不健壮的程序：直接对低级函数和数据结构编程的应用开发者必须反复地重写大量冗长乏味的软件逻辑。一般而言，编写和维护起来很乏味的代码常常含有微妙而有害的错误。

例如，在2.2.2的main函数中创建和初始化接受器socket的代码是容易出错的，比如没有对sock_addr清零，或没有对logging_port号使用htons[5]。mutex_lock和mutex_unlock也容易被误用。例如，如果write调用返回-1，logging_handler代码就会不释放互斥锁而跳出循环。同样地，如果嵌套的for循环在遇到错误时返回，socket句柄h就不会被关闭。

缺乏可移植性：使用低级函数和数据结构编写的软件常常不能在不同的OS平台和编译器间移植。而且，它们甚至常常不能在同一OS或编译器的不同版本间移植。不可移植性源于隐藏在基于低级API的函数和数据结构中的信息匮乏。

例如，在2.2.2中的日志服务器实现已经硬编码了对若干不可移植的本地OS线程和网络编程C API的依赖。特别地，对thr_create、mutex_lock和mutex_unlock的使用不能移植到非Solaris OS平台上。同样地，特定的socket特性，比如使用int表示socket句柄，不能移植到像Win32的WinSock这样的非Unix平台上；WinSock将socket句柄表示为指针。

高维护开销：C和C++开发者通常通过使用#ifdef在他们的应用源码中显式地增加条件编译指令来获得可移植性。但是，使用条件编译来处理平台特有的变种*在各方面*都增加了应用源码的*物理设计*复杂性[6]。开发者难以维护和扩展这样的软件，因为平台特有的实现细节分散在应用源文件的各个地方。

例如，处理socket数据类型的Win32和UNIX可移植性（也就是，SOCKET vs. int）的#ifdef妨碍了代码的可读性。对像这样的低级C API进行编程的开发者必须十分熟悉许多OS平台特性，才能编写和维护该代码。

由于有这些缺点，通过对低级函数和数据结构直接进行编程来开发应用常常并非是有效的设计选择。

2.2.5 解决方案

确保应用不直接访问低级函数和数据结构的一种有效途径是使用*包装外观*模式。对于每一组相关的函数和数据结构，创建一或多个包装外观类，在包装外观接口所提供的更为简洁、健壮、可移植和可维护的方法中封装低级函数和数据结构。

2.2.6 结构

包装外观模式的参与者的结构在图2-1中的UML类图中演示：

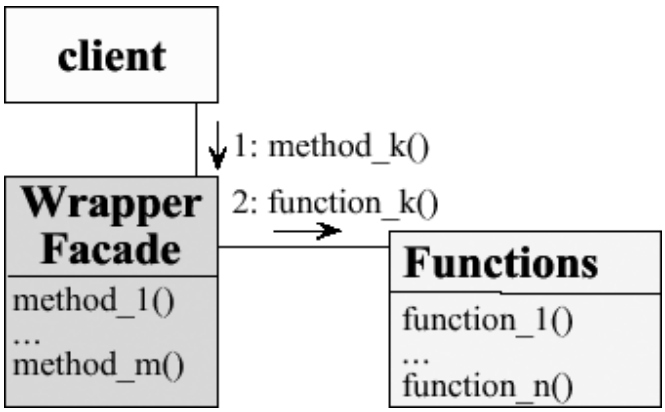

图2-1 包装外观模式的参与者的结构

包装外观模式中的关键参与者包括：

函数（Function）：*函数*是现有的低级函数和数据结构，它们提供内聚的（cohesive）服务。

包装外观（Wrapper Fa?ade）：*包装外观*是封装函数和与其相关联的数据结构的一个或一组类。包装外观提供的方法将客户调用转发给一或多个低级函数。

2.2.7 动力特性

图2-2演示包装外观模式中的各种协作：

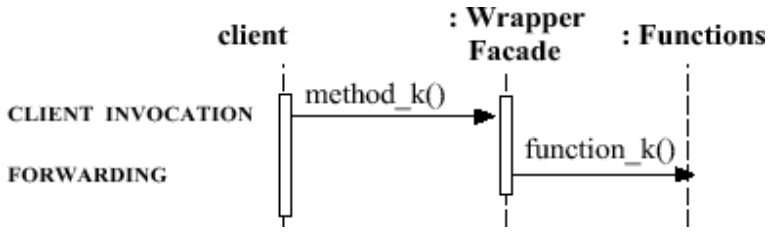

图2-2 包装外观模式中的协作

如下所述，这些协作是十分简单明了的：

1. **客户调用（**Client invocation**）：**客户通过包装外观的实例来调用方法。
2. **转发（**Forwarding**）：**包装外观方法将请求转发给它封装的一或多个底层函数，并传递函数所需的任何内部数据结构。

2.2.8 实现

这一部分解释通过包装外观模式实现组件和应用所涉及的步骤。我们将阐释这些包装外观是怎样克服繁琐、不健壮的程序、缺乏可移植性，以及高维护开销等问题的；这些问题折磨着使用低级函数和数据结构的解决方案。

这里介绍的例子基于2.2.2描述的日志服务器，图2-3演示此例中的结构和参与者。这一部分中的例子应用了来自ACE构架[7]的可复用组件。ACE提供一组丰富的可复用C++包装和构架组件，以跨越广泛的OS平台完成常见的通信软件任务。

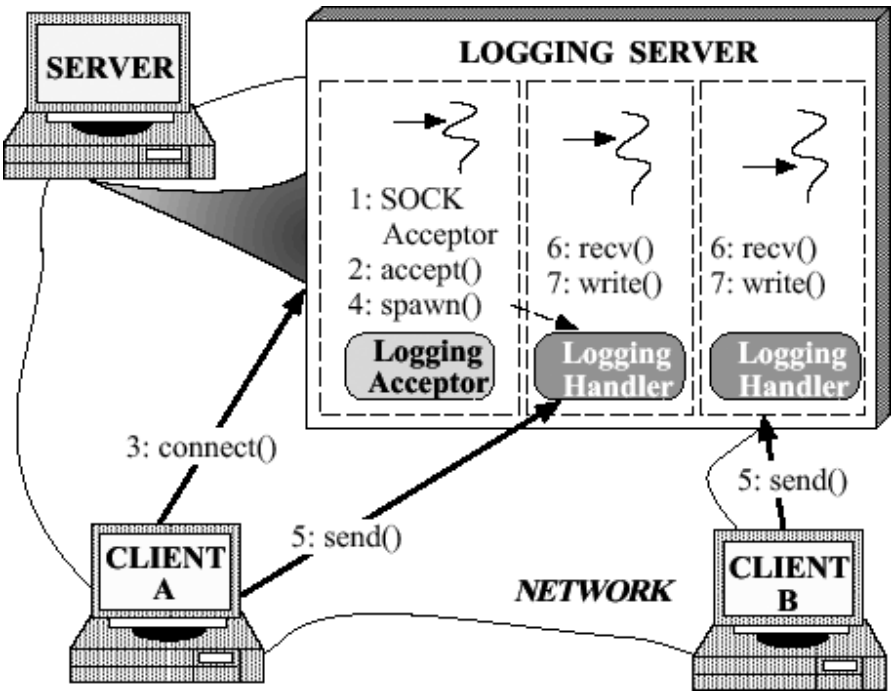

图2-3 多线程日志服务器

可采取下面的步骤来实现包装外观模式：

1. **确定现有函数间的内聚的抽象和关系：**像Win32、POSIX或X Windows这样被实现为独立的函数和数据结构的传统API提供许多内聚的抽象，比如用于网络编程、同步和线程，以及GUI管理的机制。但是，由于在像C这样的低级语言中缺乏数据抽象支持，开发者常常并不能马上明了这些现有的函数和数据结构是怎样互相关联的。因此，应用包装外观的第一步就是确定现有API中的较低级函数之间的内聚的抽象和关系。换句话说，我们通过将现有的低级API函数和数据结构聚合进一或多个类中来定义一种“对象模型”。

   在我们的日志例子中，我们从仔细检查我们原来的日志服务器实现开始。该实现使用了许多低级函数，由它们实际提供若干内聚的服务，比如同步和网络通信。例如，mutex_lock和mutex_unlock函数与互斥体同步抽象相关联。同样地，socket、bind、listen和accept函数扮演了网络编程抽象的多种角色。

2. **将内聚的函数组聚合进包装外观类和方法中：**该步骤可划分为以下子步骤：

   在此步骤中，我们为每组相关于特定抽象的函数和数据结构定义一或多个包装外观类。

   A. **创建内聚的类：**我们从为每组相关于特定抽象的函数和数据结构定义一或多个包装外观类开始。用于创建内聚的类的若干常用标准包括：

- 将具有高*内聚性*（cohesion）的函数合并进独立的类中，同时使类之间不必要的*耦合*最小化。
  - 确定在底层函数中什么是*通用的*什么是*可变的*，并把函数分组进类中，从而将变化隔离在统一的接口后面。

一般而言，如果原来的API含有广泛的相关函数，就有可能必须创建若干包装外观类来适当地对事务进行分理。

B. **将多个独立函数合并进类方法中：**除了将现有函数分组进类中，在每个包装类中将多个独立函数组合进数目更少的方法中常常也是有益的。例如，为确保一组低级函数以适当的顺序被调用，可能必须要采用此设计。

C. **选择间接层次：**大多数包装外观类简单地将它们的方法调用直接转发给底层的低级函数。如果包装外观方法是内联的，与直接调用低级函数相比，可能并没有额外的间接层次。为增强可扩展性，还可以通过动态分派包装外观方法实现来增加另外的间接层次。在这种情况下，包装外观类扮演桥接（Bridge）模式[8]中的*抽象*（Abstraction）角色。

D. **确定在哪里处理平台特有的变种：**使平台特有的应用代码最少化是使用包装外观模式的重要好处。因而，尽管包装外观类方法的*实现*在不同的OS平台上可以不同，它们应该提供统一的、平台无关的*接口*。

处理平台特有变种的一种策略是在包装外观类方法实现中使用#ifdef。在联合使用#ifdef和自动配置工具（比如GNU autoconf）时，可以通过单一的源码树创建统一的、不依赖于平台的包装外观。另一种可选策略是将不同的包装外观类实现分解进分离的目录中（例如，每个平台有一个目录），并配置语言处理工具，以在编译时将适当的包装外观类包含进应用中。

选择特定的策略在很大程度上取决于包装外观方法实现变动的频度。例如，如果它们频繁变动，为每个平台正确地更新#ifdef可能是单调乏味的。同样地，所有依赖于该文件的文件可能都需要重编译，即使变动仅仅对一个平台来说是必需的。

在我们的日志例子中，我们将为互斥体、socket和线程定义包装外观类，以演示每一子步骤是怎样被实施的。如下所示：

- **互斥体包装外观：**我们首先定义Thread_Mutex抽象，在统一和可移植的类接口中封装Solaris互斥体函数：

```
class Thread_Mutex

{

public:

Thread_Mutex (void)

{

       mutex_init (&mutex_, 0, 0);

}
```

```
?Thread_Mutex (void)
{
       mutex_destroy (&mutex_);
}

int acquire (void)
{
       return mutex_lock (&mutex_);
}

int release (void)
{
       return mutex_unlock (&mutex_);
}

private:
// Solaris-specific Mutex mechanism.
mutex_t mutex_;

// = Disallow copying and assignment.
Thread_Mutex (const Thread_Mutex &);
void operator= (const Thread_Mutex &);
};
```

通过定义Thread_Mutex类接口，并随之编写使用它、而不是低级本地OS C API的应用，我们可以很容易地将我们的包装外观移植到其他平台。例如，下面的Thread_Mutex实现在Win32上工作：

```
class Thread_Mutex
{
public:
Thread_Mutex (void)
{
       InitializeCriticalSection (&mutex_);
}
```

```
?Thread_Mutex (void)

{

       DeleteCriticalSection (&mutex_);

}



int acquire (void)

{

       EnterCriticalSection (&mutex_); return 0;

}



int release (void)

{

       LeaveCriticalSection (&mutex_); return 0;

}



private:

// Win32-specific Mutex mechanism.

CRITICAL_SECTION mutex_;



// = Disallow copying and assignment.

Thread_Mutex (const Thread_Mutex &);

void operator= (const Thread_Mutex &);

};
```

如早先所描述的，我们可以通过在Thread_Mutex方法实现中使用#ifdef以及自动配置工具（比如GUN autoconf）来支持多个OS平台，以使用单一源码树提供统一的、平台无关的互斥体抽象。相反，我们也可以将不同的Thread_Mutex实现分解进分离的目录中，并指示我们的语言处理工具在编译时将适当的版本包含进我们的应用中。

除了改善可移植性，我们的Thread_Mutex包装外观还提供比直接编程低级Solaris函数和mutex_t数据结构更不容易出错的互斥体接口。例如，我们可以使用C++ private访问控制指示符来禁止互斥体的拷贝和赋值；这样的使用是错误的，但却不会被不那么强类型化的C编程API所阻止。

- **socket包装外观：**socket API比Solaris互斥体API要大得多，也有表现力得多[5]。因此，我们必须定义一组相关的包装外观类来封装socket。我们将从定义下面的处理UNIX/Win32可移植性差异的typedef开始：

```
#if !defined (_WINSOCKAPI_)
```

```
typedef int SOCKET;

#define INVALID_HANDLE_VALUE -1

#endif /* _WINSOCKAPI_ */
```

接下来，我们将定义INET_Addr类，封装Internet域地址结构：

```
class INET_Addr

{

public:

INET_Addr (u_short port, long addr)

{

       // Set up the address to become a server.

       memset (reinterpret_cast <void *> (&addr_), 0, sizeof addr_);

       addr_.sin_family = AF_INET;

       addr_.sin_port = htons (port);

       addr_.sin_addr.s_addr = htonl (addr);

}


u_short get_port (void) const

{

       return addr_.sin_port;

}


long get_ip_addr (void) const

{

       return addr_.sin_addr.s_addr;

}


sockaddr *addr (void) const

{

       return reinterpret_cast <sockaddr *>(&addr_);

}


size_t size (void) const

{
```

```
        return sizeof (addr_);
}
// ...

private:
sockaddr_in addr_;
};
```

注意INET_Addr构造器是怎样通过将sockaddr_in域清零，并确保端口和IP地址被转换为网络字节序，消除若干常见的socket编程错误的。

下一个包装外观类，SOCK_Stream，对应用可在已连接socket句柄上调用的I/O操作（比如recv和send）进行封装：

```
class SOCK_Stream
{
public:
// = Constructors.
// Default constructor.
SOCK_Stream (void)
        : handle_ (INVALID_HANDLE_VALUE) {}

// Initialize from an existing HANDLE.
SOCK_Stream (SOCKET h): handle_ (h) {}

// Automatically close the handle on destruction.
?SOCK_Stream (void) { close (handle_); }

void set_handle (SOCKET h) { handle_ = h; }
SOCKET get_handle (void) const { return handle_; }

// = I/O operations.
int recv (char *buf, size_t len, int flags = 0);
int send (const char *buf, size_t len, int flags = 0);
// ...

private:
```

```
// Handle for exchanging socket data.

SOCKET handle_;

};
```

注意此类是怎样确保socket句柄在SOCK_Stream对象出作用域时被自动关闭的。

SOCK_Stream对象由连接工厂SOCK_Acceptor创建，后者封装*被动的*连接建立逻辑[9]。SOCK_Acceptor构造器初始化被动模式接受器socket，以在sock_addr地址上进行侦听。同样地，accept工厂方法通过新接受的连接来初始化SOCK_Stream，如下所示：

```
class SOCK_Acceptor

{

public:

SOCK_Acceptor (const INET_Addr &sock_addr)

{

       // Create a local endpoint of communication.

       handle_ = socket (PF_INET, SOCK_STREAM, 0);


       // Associate address with endpoint.

       bind (handle_, sock_addr.addr (), sock_addr.size ());


       // Make endpoint listen for connections.

       listen (handle_, 5);

};


// Accept a connection and initialize

// the <stream>.

int accept (SOCK_Stream &stream)

{

       stream.set_handle (accept (handle_, 0, 0));

       if (stream.get_handle () == INVALID_HANDLE_VALUE)

              return -1;

       else return 0;

}


private:

// Socket handle factory.
```

```
SOCKET handle_;

};
```

注意SOCK_Acceptor的构造器是怎样确保低级的socket、bind和listen函数总是以正确的次序被调用的。

完整的socket包装外观集还包括SOCK_Connector，封装*主动的*连接建立逻辑[9]。

- **线程外观：**在不同的OS平台上有许多线程API可用，包括Solaris线程、POSIX Pthreads和Win32线程。这些API显示出微妙的语法和语义差异，例如，Solaris和POSIX线程可以“分离”（detached）模式被派生，而Win32线程则不行。但是，可以提供Thread_Manager包装外观，在统一的API中封装这些差异。如下所示：

```
class Thread_Manager

{

public:

int spawn (void *(*entry_point) (void *),

             void *arg,

                    long flags,

                    long stack_size = 0,

                    void *stack_pointer = 0,

                    thread_t *t_id = 0)

{

       thread_t t;

       if (t_id == 0)

              t_id = &t;


       return thr_create (stack_size,

                                        stack_pointer,

                                        entry_point,

                                        arg,

                                        flags,

                                        t_id);

}

// ...

};
```

Thread_Manager还提供联接（join）和取消线程的方法。

1. **确定错误处理机制：**低级的C函数API通常使用返回值和整型代码（比如errno）来将错误通知给它们的调用者。但是，此技术是容易出错的，因为调用者可能会忘记检查它们的函数调用的返回状态。

   更为优雅的报告错误的方式是使用异常处理。许多编程语言，比如C++和Java，使用异常处理来作为错误报告机制。它也被某些操作系统所使用，比如Win32。

   使用异常处理作为包装外观类的错误处理机制有若干好处：

- **它是可扩展的：**现代编程语言允许通过对现有接口和使用干扰极少的特性来扩展异常处理策略和机制。例如，C++和Java使用继承来定义异常类的层次。
- **它使错误处理与正常处理得以干净地去耦合：**例如，错误处理信息不会显式地传递给操作。而且，应用不会因为没有检查函数返回值而偶然地忽略异常。
- **它可以是类型安全的：**在像C++和Java这样的语言中，异常以一种强类型化的方式被扔出和捕捉，以增强错误处理代码的组织和正确性。相对于显式地检查线程专有的错误值，编译器会确保对于每种类型的异常，将执行正确的处理器。

但是，为包装外观类使用异常处理也有若干缺点：

- **它不是通用的：**不是所有语言都提供异常处理。例如，某些C++编译器没有实现异常。同样地，当OS提供异常服务时，它们必须被语言扩展所支持，从而降低了代码的可移植性。
- **它使多种语言的使用变得复杂化：**因为语言以不同的方式实现异常，或根本不实现异常，如果以不同语言编写的组件扔出异常，可能很难把它们集成在一起。相反，使用整型值或结构来报告错误信息提供了更为通用的解决方案。
- **它使资源管理变得复杂化：**如果在C++或Java代码块中有多个退出路径，资源管理可能会变得复杂化[10]。因而，如果语言或编程环境不支持垃圾收集，必须注意确保在有异常扔出时删除动态分配的对象。
- **它有着潜在的时间和/或空间低效的可能性：**即使没有异常扔出，异常处理的糟糕实现也会带来时间和/或空间的过度开销[10]。对于必须具有高效和低内存占用特性的嵌入式系统来说，这样的开销可能会特别地成问题。

对于封装内核级设备驱动程序或低级的本地OS API（它们必须被移植到许多平台上）的包装外观来说，异常处理的缺点也是特别成问题的。对于这些类型的包装外观，更为可移植、高效和线程安全的处理错误的方式是定义错误处理器抽象，显式地维护关于操作的成功或失败的信息。使用线程专有存储（Thread-Specific Storage）模式[11]是被广泛用于这些系统级包装外观的解决方案。

1. **定义相关助手类（可选）：**一旦低级函数和数据结构被封装在内聚的包装外观类中，常常有可能创建其他助手类来进一步简化应用开发。通常要在包装外观模式已被应用于将低级函数和与其关联的数据聚合进类中之后，这些助手类的效用才变得明显起来。

例如，在我们的日志例子中，我们可以有效地利用下面的实现C++ *Scoped Locking*习语的Guard类；该习语确保Thread_Mutex被适当地释放，不管程序的控制流是怎样退出作用域的。

```
template <class LOCK>
class Guard
{
public:
Guard (LOCK &lock): lock_ (lock)
{
       lock_.acquire ();
}

?Guard (void)
{
       lock_.release ();
}

private:
// Hold the lock by reference to avoid
// the use of the copy constructor...
LOCK &lock_;
}
```

Guard类应用了[12]中描述的C++习语，藉此，在一定作用域中“构造器获取资源而析构器释放它们”。如下所示：

```
// ...
{
// Constructor of <mon> automatically
// acquires the <mutex> lock.
Guard<Thread_Mutex> mon (mutex);

// ... operations that must be serialized ...

// Destructor of <mon> automatically
```

```
// releases the <mutex> lock.

}

// ...
```

因为我们使用了像Thread_Mutex包装外观这样的*类*，我们可以很容易地替换不同类型的锁定机制，与此同时仍然复用Guard的自动锁定/解锁协议。例如，我们可以用Process_Mutex类来取代Thread_Mutex类，如下所示：

```
// Acquire a process-wide mutex.

Guard<Process_Mutex> mon (mutex);
```

如果使用C函数和数据结构、而不是C++类，获得这种程度的“可插性”（pluggability）要困难得多。

2.2.9 例子解答

下面的代码演示日志服务器的main函数，它已使用2.2.8描述的互斥体、socket和线程的包装外观重写。

```
// At file scope.

// Keep track of number of logging requests.

static int request_count;


// Manage threads in this process.

static Thread_Manager thr_mgr;


// Lock to protect request_count.

static Thread_Mutex lock;


// Forward declaration.

static void *logging_handler (void *);


// Port number to listen on for requests.

static const int logging_port = 10000;


// Main driver function for the multi-threaded
```

```
// logging server. Some error handling has been
// omitted to save space in the example.
int main (int argc, char *argv[])
{
// Internet address of server.
INET_Addr addr (port);

// Passive-mode acceptor object.
SOCK_Acceptor server (addr);

SOCK_Stream new_stream;

// Wait for a connection from a client.
for (;;)
{
       // Accept a connection from a client.
       server.accept (new_stream);

       // Get the underlying handle.
       SOCKET h = new_stream.get_handle ();

       // Spawn off a thread-per-connection.
       thr_mgr.spawn (logging_handler,
                                 reinterpret_cast <void *> (h),
                                 THR_DETACHED);
       }
}
```

logging_handler函数运行在单独的线程控制中，也就是，每个相连客户有一个线程。它在各个连接上接收并处理日志记录，如下所示：

```
// Entry point that processes logging records for
// one client connection.
void *logging_handler (void *arg)
{
SOCKET h = reinterpret_cast <SOCKET> (arg);
```

```
// Create a <SOCK_Stream> object from SOCKET <h>.
SOCK_Stream stream (h);
for (;;)
{
       UINT_32 len; // Ensure a 32-bit quantity.
       char log_record[LOG_RECORD_MAX];

       // The first <recv_n> reads the length
       // (stored as a 32-bit integer) of
       // adjacent logging record. This code
       // handles "short-<recv>s".
       ssize_t n = stream.recv_n
                           (reinterpret_cast <char *> (&len),
                           sizeof len);

       // Bail out if we're shutdown or
       // errors occur unexpectedly.
       if (n <= 0) break;
       len = ntohl (len); // Convert byte-ordering.
       if (len > LOG_RECORD_MAX) break;

       // The second <recv_n> then reads <len>
       // bytes to obtain the actual record.
       // This code handles "short-<recv>s".
       n = stream.recv_n (log_record, len);

       // Bail out if we're shutdown or
       // errors occur unexpectedly.
       if (n <= 0) break;

       {
              // Constructor of Guard automatically
              // acquires the lock.
              Guard<Thread_Mutex> mon (lock);
```

```
            // Execute following two statements in a

            // critical section to avoid race conditions

            // and scrambled output, respectively.

            ++request_count; // Count # of requests


            if (write (STDOUT, log_record, len) == -1)

                        break;


            // Destructor of Guard automatically

            // releases the lock, regardless of

            // how we exit this block!

      }

}


// Destructor of <stream> automatically

// closes down <h>.

return 0;

}
```

注意上面的代码是怎样解决2.2.2所示代码的各种问题的。例如，SOCK_Stream和Guard的析构器会分别关闭socket句柄和释放Thread_Mutex，而不管代码块是怎样退出的。同样地，此代码要容易移植和维护得多，因为它没有使用平台特有的API。

2.2.10 已知应用

本论文中的例子聚焦于并发网络编程。但是，包装外观模式已被应用到其他的许多领域，比如GUI构架和数据库类库。下面是包装外观模式的一些广为人知的应用：

Microsoft Foundation Class（MFC）：MFC提供一组封装大多数低级C Win32 API的包装外观，主要集中于提供实现Microsoft文档/模板体系结构的GUI组件。

ACE**构架**：2.2.8描述的互斥体、线程和socket的包装外观分别基于ACE构架中的组件[7]：ACE_Thread_Mutex、ACE_Thread_Manager和ACE_SOCK*类。

Rogue Wave**类库**：Rogue Wave的Net.h++和Threads.h++类库在许多OS平台上实现了socket、线程和同步机制的包装外观。

ObjectSpace System<Toolkit>：该工具包也提供了socket、线程和同步机制的包装外观。

Java虚拟机和Java基础类库：Java虚拟机（JVM）和各种Java基础类库，比如AWT和Swing，提供了一组封装大多数低级的本地OS系统调用和GUI API的包装外观。

2.2.11 效果

包装外观模式提供以下好处：

更为简洁和健壮的编程接口：包装外观模式在一组更为简洁的OO类方法中封装许多低级函数。这减少了使用低级函数和数据结构开发应用的枯燥性，从而降低了发生编程错误的潜在可能性。

改善应用可移植性和可维护性：包装外观类的实现可用以使应用开发者与低级函数和数据结构的不可移植的方面屏蔽开来。而且，通过用基于*逻辑设计*实体（比如基类、子类，以及它们的关系）的应用配置策略取代基于*物理设计*实体（比如文件和#ifdef）的策略[6]，包装外观模式改善了软件结构。一般而言，根据应用的逻辑设计、而不是物理设计来理解和维护它们要更为容易一些。

改善应用的模块性、可复用性和可配置性：通过使用像继承和参数化类型这样的OO语言特性，包装外观模式创建的可复用类组件可以一种整体方式被“插入”其他组件，或从中“拔出”。相反，不求助于粗粒度的OS工具，比如链接器或文件系统，替换成组的函数要难得多。

包装外观模式有以下缺点：

额外的间接性（Indirection）：与直接使用低级的函数和数据结构相比，包装外观模式可能带来额外的间接。但是，支持内联的语言，比如C++，可以无需显著的开销而实现该模式，因为编译器可以内联用于实现包装外观的方法调用。

2.2.12 参见

包装外观模式与外观模式是类似的[8]。外观模式的意图是简化子系统的接口。包装外观模式的意图则更为具体：它提供简洁、健壮、可移植和可维护的类接口，封装低级的函数和数据结构，比如本地OS互斥体、socket、线程和GUI C语言API。一般而言，外观将复杂的类关系隐藏在更简单的API后面，而包装外观将复杂的函数和数据结构关系隐藏在更丰富的类API后面。

如果动态分派被用于实现包装外观方法，包装外观模式可使用桥接模式[8]来实现；包装外观方法在桥接模式中扮演*抽象*（Abstraction）角色。

2.3 结束语

本论文描述包装外观模式，并给出了详细的例子演示怎样使用它。在本论文中描述的ACE包装外观组件的实现可在ACE[7]软件发布中自由获取（URL：http://www.cs.wustl.edu/~schmidt/ACE.html）。该发布含有在圣路易斯华盛顿大学开发的完整的C++源码、文档和测试例子驱动程序。目前ACE正在用于许多公司（像Bellcore、波音、DEC、爱立信、柯达、朗讯、摩托罗拉、SAIC和西门子）的通信软件项目中。

感谢

感谢Hans Rohnert、Regine Meunier、Michael Stal、Christa Schwanninger、Frank Buschmann和Brad Appleton，他们的大量意见极大地改善了包装外观模式描述的形式和内容。

参考文献

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[2] W.R.Stevens, *UNIX Network Programming, First Edition*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[3] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalin-giah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Ker-nel," in *Proceedings of the Summer USENIX Conference*,(San Antonio, Texas), June 1992.

[4] W.R.Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[5] D. C. Schmidt, "IPC SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*,vol.4, November/December 1992.

[6] J. Lakos, *Large-scale Software Development with C++*. Reading, MA: Addison-Wesley, 1995.

[7] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Mas-sachusetts), USENIX Association, April 1994.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pat-terns: Elements of Reusable Object-Oriented Softwar*e. Read-ing, MA: Addison-Wesley, 1995.

[9] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[10] H. Mueller, "Patterns for Handling Exception Handling Suc-cessfully," *C++ Report*, vol. 8, Jan. 1996.

[11] D. C. Schmidt, T. Harrison, and N. Pryce, "Thread-Specific Storage - An Object Behavioral Pattern for Accessing per-Thread State Efficiently," *C++ Report*,vol.9,Novem-ber/December 1997.

[12] Bjarne Stroustrup, *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1998.