第9章 接受器－连接器（Acceptor-Connector）：用于连接和初始化通信服务的对象创建模式

Douglas C. Schmidt

9.1 意图

接受器－连接器设计模式（Acceptor-Connector）使分布式系统中的连接建立及服务初始化与一旦服务初始化后所执行的处理去耦合。这样的去耦合通过三种组件来完成：*acceptor*、*connector*和*service handler*（服务处理器）。连接器*主动地*建立到远地接受器组件的连接，并初始化服务处理器来处理在连接上交换的数据。同样地，接受器*被动地*等待来自远地连接器的连接请求，在这样的请求到达时建立连接，并初始化服务处理器来处理在连接上交换的数据。随后已初始化的服务处理器执行应用特有的处理，并通过连接器和接受器组件建立的连接来进行通信。

9.2 例子

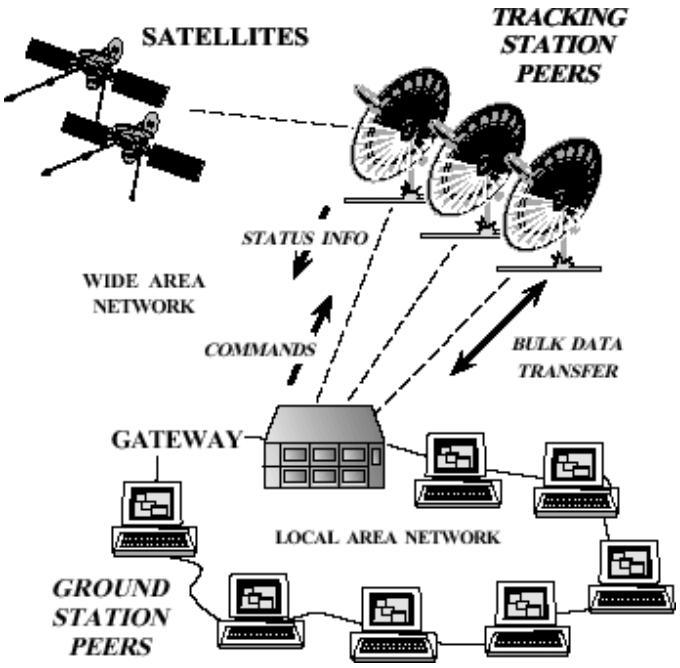

图9-1 面向连接的应用级网关的物理体系

为演示接受器 - 连接器模式，考虑图9-1中所示的多服务、应用级Gateway（网关）。通常，Gateway使分布式系统中的协作组件去耦合，并允许它们在无需互相直接依赖的情况下进行交互。图9-1中的这个Gateway在不同的服务端点间路由数据，这些端点运行在用于监视和控制人造卫星群的远地Peer（对端）上。每个Peer中的服务经由Gateway发送和接收若干类型的数据,比如状态信息、大块数据和命令。一般而言，Peer可以分布在局域网和广域网中。

该Gateway是一个路由器，负责协调它的Peer之间的通信。从Gateway的角度来看，它为之路由数据的Peer服务仅由其应用级通信协议来进行区分，这些协议可能会使用不同的帧格式和有效负载类型。

Gateway在它的Peer之间使用面向连接的TCP/IP协议[11]来传输数据。在我们的示例网络配置中，每个服务都与一个连接端点绑定在一起；该端点由IP主机地址和TCP端口号指定。端口号唯一地标识服务的类型。为每种类型的服务/端口维护单独的连接增强了路由策略的灵活性，并提供了更为健壮的错误处理，如果网络连接意外关闭的话。

在我们的分布式应用中，Gateway和Peer必须能改变它们的连接角色，以支持不同的使用情况。特别地，或是可以主动地发起连接，或是可以被动地等待连接请求。例如，在一种配置中，Gateway可以主动地发起到远地Peer的连接，以便路由数据给它们。在另一种配置中，Gateway可以被动地接收来自Peer的连接请求，后者随即经由Gateway路由数据给另外的Peer。同样地，在一种使用情况下，Peer可以是主动的连接发起者，而在另一种使用情况下则是被动的连接接受者。

由于我们的分布式应用的天性，预先指定连接建立和服务初始化角色、并将它们硬编码进Gateway和Peer组件的传统设计太不灵活。这样的设计过度地将连接建立、服务初始化和服务处理组件耦合在一起。这样的紧耦合使得独立于通信角色改变连接角色变得很困难。

9.3 上下文

分布式系统中利用面向连接协议来在服务端点间进行通信的客户/服务器应用。

9.4 问题

分布式应用常常含有复杂的代码，执行连接建立和服务初始化。一般而言，分布式应用中在服务端点间交换的数据的处理极大地独立于配置问题，比如（1）是哪一个端点发起连接，也就是，*连接角色* vs. *通信角色*，以及（2）连接管理协议 vs. 网络编程API。下面对这些问题进行概述：

- **连接角色** vs. **通信角色**：连接建立角色天然地*不对称*，也就是，被动的服务端点进行*等待*，而主动的服务端点*发起*连接。但是，一旦连接建立，通信角色和连接角色可以是互不相关的。因而，数据可以任意的服从服务通信协议的方式在服务端点间进行传输。常用的通信协议包括点对点、请求 - 响应和单路流。
- **连接管理协议** vs. **网络编程**API：不同的网络编程接口，比如socket或TLI，提供不同的API来使用多种连接管理协议建立连接。但是，不管用于建立连接的协议是什么，数据可以使用统一的消息传递操作来在端点间进行传输，例如，send/recv调用。

一般而言，用于连接建立和服务初始化的策略变动的频度要远小于应用服务实现和通信协议。因而，使这些方面去耦合、以使它们能独立地变化，对于开发和维护分布式应用来说是必要的。对于将连接及初始化协议与通信协议分离的问题，下面一些*压力*会对解决方案产生影响：

- 应该容易增加新类型的服务、新的服务实现和新的通信协议，而又不影响现有的连接建立和服务初始化软件。例如，可能有必要扩展Gateway，以与运行在IPX/SPX通信协议、而不是TCP/IP之上的目录服务进行互操作。
- 应该有可能使下面的两种角色去耦合：（1）*连接角色*，也就是，哪一个进程发起连接 vs. 接受连接，以及（2）*通信角色*，也就是，哪一个服务端点是客户或服务器。通常，“客户”和“服务器”之间的区分指的是通信角色，它们可以与连接角色不相关。例如，在发起到被动服务器的连接时，客户常常扮演主动角色。但是，这些连接角色可以反转过来。例如，扮演主动通信角色的客户可以被动地等待另一个进程对其进行连接。9.2中的例子演示了后一种使用情况。
- 应该有可能编写可以移植到许多OS平台上的通信软件，以最大化可用性和市场占有率。许多低级网络编程API的语义只是有着表面的不同，而语法却互不兼容，因而难以使用低级API，比如socket和TLI，来编写可移植应用。
- 应该有可能将程序员与低级网络编程API（像socket或TLI）类型安全性的缺乏屏蔽开来。例如，连接建立代码应完全地与后续的数据传输代码去耦合，以确保端点被正确地使用。没有这种强去耦，服务可能会错误地在被动模式的传输端点工厂上读写数据，而后者仅应被用于接受连接。
- 应该有可能通过使用像异步连接建立这样的OS特性来降低连接响应延迟。例如，有大量对端的应用可能需要异步、并发地建立许多连接。高效和可伸缩的连接建立对于运行在高响应延迟的WAN上的应用来说特别重要。
- 应该可以尽可能多地复用通用的连接建立和服务初始化软件，以有效利用先前的开发成果。

9.5 解决方案

对于分布式应用提供的每个*服务*，使用*接受器 - 连接器*模式来使连接建立及服务初始化与由服务的两个端点在连接和初始化之后执行的后续处理去耦合。

引入两个工厂，生成已连接和初始化的服务处理器，用于实现应用的服务。第一个工厂，称为接受器，创建并初始化传输端点，被动地在特定地址上侦听来自远地连接器的连接请求。第二个工厂，连接器，主动地发起到远地接受器的连接。接受器和连接器都初始化相应的服务处理器，处理在连接上交换的数据。一旦服务处理器被连接和初始化，它们就执行应用特有的处理，一般不再与接受器和连接器进行交互。

9.6 结构

在图9-2中通过Booch类图[2]演示了接受器 - 连接器模式中的参与者的结构。

服务处理器（Service Handler）：Service Handler实现应用服务，通常扮演客户角色、服务器角色，或同时扮演这两种角色。它提供挂钩方法，由Acceptor或Connector调用，以在连接建立时启用应用服务。此外，Service Handler还提供数据模式传输端点，其中封装了一个I/O句柄，比如socket。一旦连接和初始化后，该端点被Service Handler用于与和其相连的对端交换数据。

接受器（Acceptor）：Acceptor是一个工厂，实现用于*被动地*建立连接并初始化与其相关联的Service Handler的策略。此外，Acceptor包含有被动模式的传输端点工厂，它创建新的数据模式端点，由Service Handler用于在相连的对端间传输数据。通过将传输端点工厂绑定到网络地址，比如Acceptor在其上侦听的TCP端口号，Acceptor的open方法对该工厂进行初始化。

一旦初始化后，被动模式的传输端点工厂侦听来自对端的连接请求。当连接请求到达时，Acceptor创建Service Handler，并使用它的传输端点工厂来将新连接接受进Service Handler中。

连接器（Connector）：Connector是一个工厂，实现用于*主动地*建立连接并初始化与其相关联的Service Handler的策略。它提供方法，由其发起到远地Acceptor的连接。同样地，它还提供另一个方法，完成对Service Handler的启用；该处理器的连接是被同步或异步地发起的。Connector使用两个分开的方法来透明地支持异步连接建立。

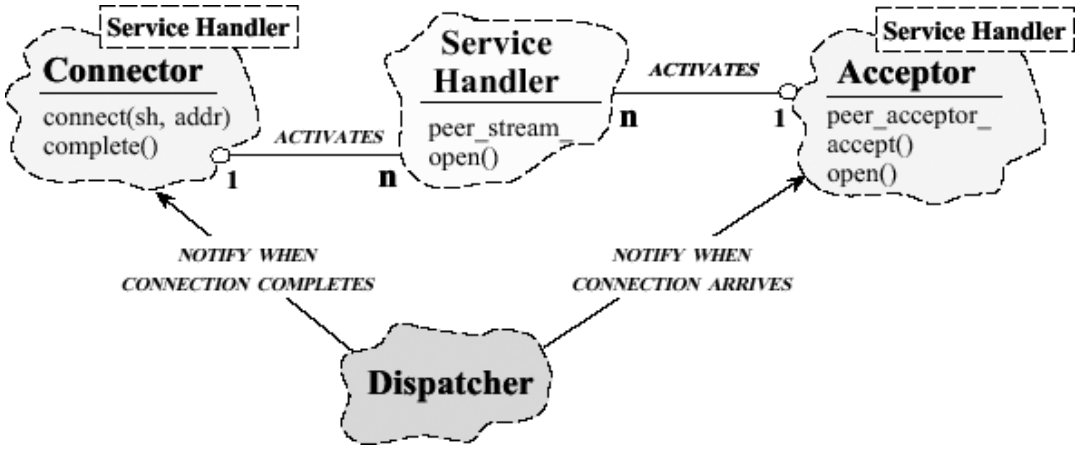

图9-2 Acceptor-Connector模式的参与者的结构

当连接建立时，Acceptor和Connector都通过调用Service Handler的启用挂钩方法来将其启用。一旦Service Handler被Acceptor或Connector工厂完全初始化，它通常就不再与这些组件进行交互了。

分派器（Dispatcher）：为Acceptor，Dispatcher将在一或多个传输端点上接收到的连接请求多路分离给适当的Acceptor。Dispatcher允许多个Acceptor向其登记，以侦听同时在不同端口上从不同对端而来的连接。

为Connector，Dispatcher处理异步发起的连接的完成。在这种情况下，当异步连接被建立时，Dispatcher回调Connector。Dispatcher允许多个Service Handler通过一个Connector来异步地发起和完成它们的连接。注意对于同步连接建立，Dispatcher并不是必需的，因为发起连接的线程控制也完成服务服务处理器的启用。

Dispatcher通常使用事件多路分离模式来实现，这些模式由反应堆（Reactor）[3]或前摄器（Proactor）[4]来提供，它们分别处理同步和异步的多路分离。同样地，Dispatcher也可以使用主动对象（Active Object）模式[5]来实现为单独的线程或进程。

9.7 动力特性

下面的部分描述接受器 - 连接器模式中Acceptor和Connector组件所执行的协作。我们检查三种规范的情况：Acceptor、异步的Connector和同步的Connector。

9.7.1 Acceptor组件协作

图9-3演示Acceptor和Service Handler之间的协作。这些协作被划分为三个阶段：

1. **端点初始化阶段：**为被动地初始化连接，应用调用Acceptor的open方法。该方法创建被动模式的传输端点，将其绑定到网络地址，例如，本地主机的IP地址和TCP端口号，并随后侦听来自对端Connector的连接请求。其次，open方法将Acceptor对象登记到Dispatcher，以使分派器能够在连接事件到达时回调Acceptor。最后，应用发起Dispatcher的事件循环，等待连接请求从对端Connector到来。
2. **服务初始化阶段：**当连接请求到达时，Dispatcher回调Acceptor的accept方法。该方法装配以下活动所必需的资源：（1）创建新的Service Handler，（2）使用它的被动模式传输端点工厂来将连接接受进该处理器的数据模式传输端点中，以及（3）通过调用Service Handler的open挂钩将其启用。Service Handler的open挂钩可以执行服务特有的初始化，比如分配锁、派生线程、打开日志文件，和/或将该Service Handler登记到Dispatcher。
3. **服务处理阶段：**在连接被动地建立和Service Handler被初始化后，服务处理阶段开始了。在此阶段，应用级通信协议，比如HTTP或IIOP，被用于在本地Service Handler和与其相连的远地Peer之间、经由前者的peer_stream_端点交换数据。当交换完成，可关闭连接和Service Handler，并释放资源。

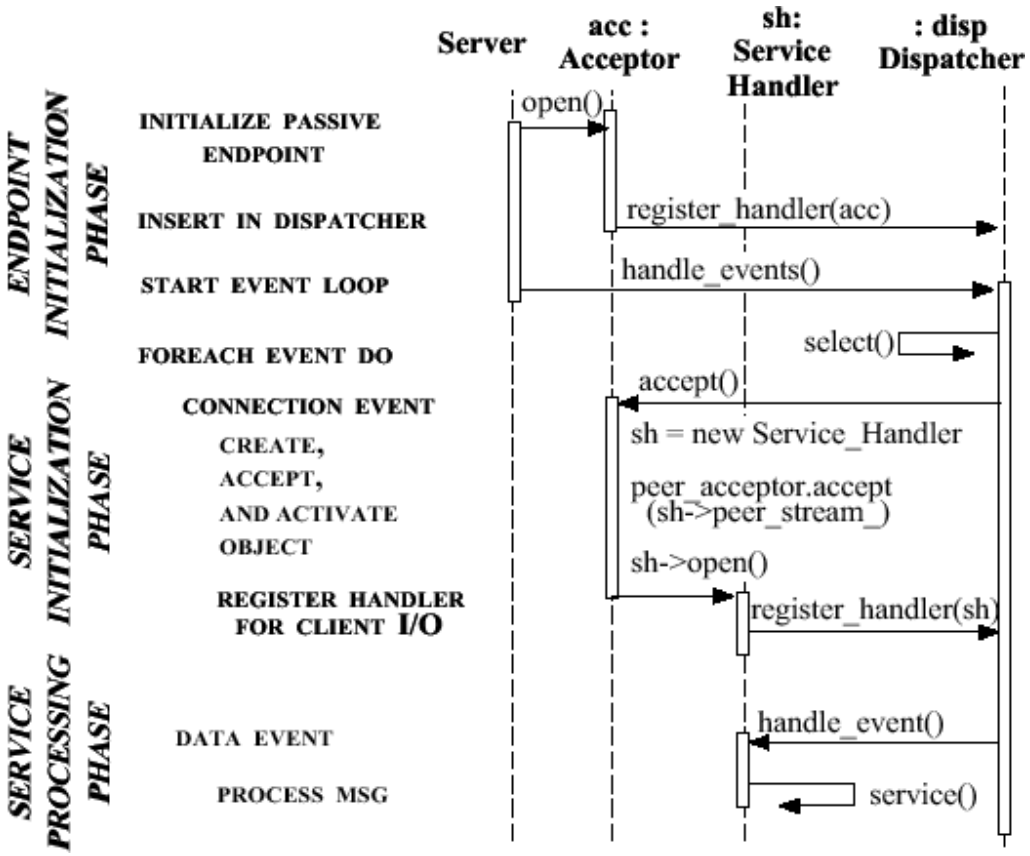

图9-3 Acceptor参与者之间的协作

9.7.2 Connector组件协作

Connector组件可以使用两种常用方案来初始化它的Service Handler：*同步的*和*异步的*。同步的服务初始化对于以下情形来说是有用的：

- 如果建立连接的延迟非常低，例如，经由回路设备与在同一主机上的服务器建立连接；或是
- 如果有多个线程控制可用，并且使用不同的线程来同步地连接每个Service Handler有足够的效率；或是
- 如果服务必须以固定顺序初始化，而客户不到连接建立不能执行其他有用的工作。

同样地，异步服务初始化在相反的情形中是有用的：

- 如果连接延迟很高，并且有许多对端需要连接，例如，在高延迟WAN之上建立大量连接；或是
- 如果仅有单个线程控制可用，例如，如果OS平台不提供应用级线程；或是
- 如果服务被初始化的顺序不重要，或如果客户应用必须在建立连接的同时执行另外的工作，比如刷新GUI。

*同步的*Connector情况中的参与者之间的协作可被划分为以下三个阶段：

1. **连接发起阶段：**为在Service Handler和它的远地Peer之间发起连接，应用调用Connector的connect方法。该方法阻塞调用线程的线程控制、直到连接同步完成，以主动地建立连接。
2. **服务初始化阶段：**在连接完成后，Connector的connect方法调用complete方法来启用Service Handler。complete方法通过调用Service_Handler的open挂钩方法来完成启用；open方法执行服务特有的初始化。
3. **服务处理阶段：**此阶段与Service Handler被Acceptor创建后所执行的服务处理阶段相类似。特别地，一旦Service Handler被启用，它使用与和其相连接的远地Service Handler交换的数据来执行应用特有的服务处理。

同步服务初始化的协作如图9-4所示。在此方案中，Connector将连接发起和服务初始化阶段结合进单一的阻塞操作中。在此情况中，只为每个线程控制中的每次connect调用建立一个连接。

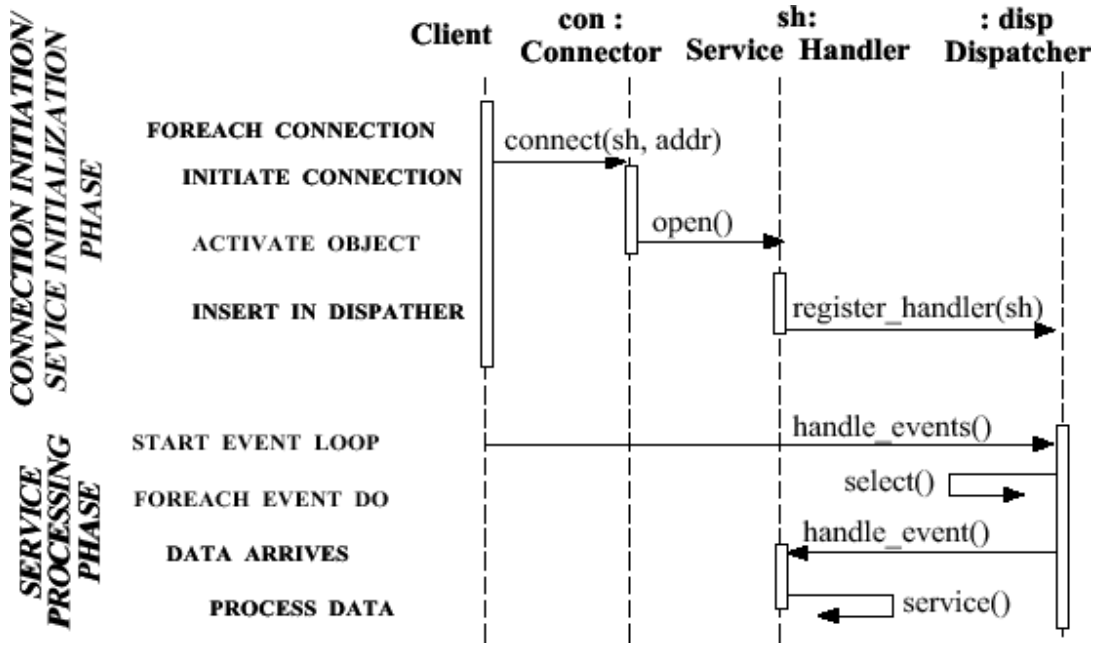

图9-4 用于同步连接的Connector参与者之间的协作

*异步的*Connector中的参与者之间的协作可被划分为以下三个阶段：

1. **连接发起阶段：**为在Service Handler和其远地Peer之间发起一个连接，应用调用Connector的connect方法。就如同同步方案，Connector主动地建立连接。但是，在连接异步完成的同时，它不会阻塞调用者的线程控制。相反，它将Service Handler的传输端点（我们在此例中将其称为peer_stream_）登记到Dispatcher，并将控制返回给它的调用者。
2. **服务初始化阶段：**在连接异步完成后，Dispatcher回调Connector的complete方法。该方法通过调用Service Handler的open挂钩来将其启用。这个open挂钩执行服务特有的初始化。
3. **服务处理阶段：**此阶段与前面描述的其他服务处理阶段相类似。一旦Service Handler被启用，它使用与和其相连接的远地Service Handler交换的数据来执行应用特有的服务处理。

图9-5演示这三个阶段的使用异步连接建立的协作。在异步方案中，注意连接发起阶段被暂时与服务初始化阶段分离开来。这样的去耦合使得多个连接发起（经由connect）和完成（经由complete）能够在各自的线程控制中并行地进行。

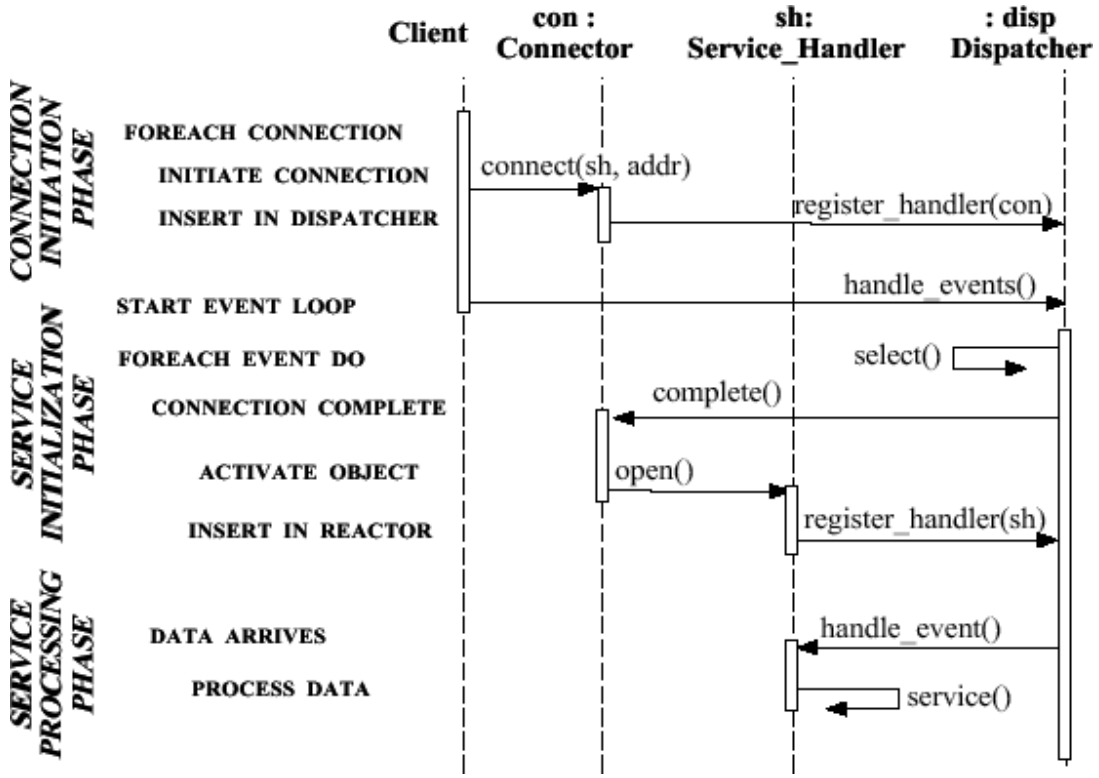

图9-5 用于异步连接的Connector参与者之间的协作

9.8 实现

这一部分解释使用接受器－连接器模式来构建通信软件应用所涉及的步骤。这里的实现基于ACE OO网络编程工具包[6]中的可复用组件和应用。ACE提供一组丰富的可复用C++包装和构架组件，它们可在一系列OS平台上执行常用的通信软件任务。

接受器－连接器模式中的参与者被划分为*反应*、*连接*和*应用*层，如图9-6所示。

反应和连接层分别为分派事件和初始化服务执行通用的、与应用无关的策略。应用层通过提供建立连接和执行服务处理的具体类来实例化这些通用策略。这样的事务分离增强了接受器－连接器模式实现中的可复用性、可移植性和可扩展性。

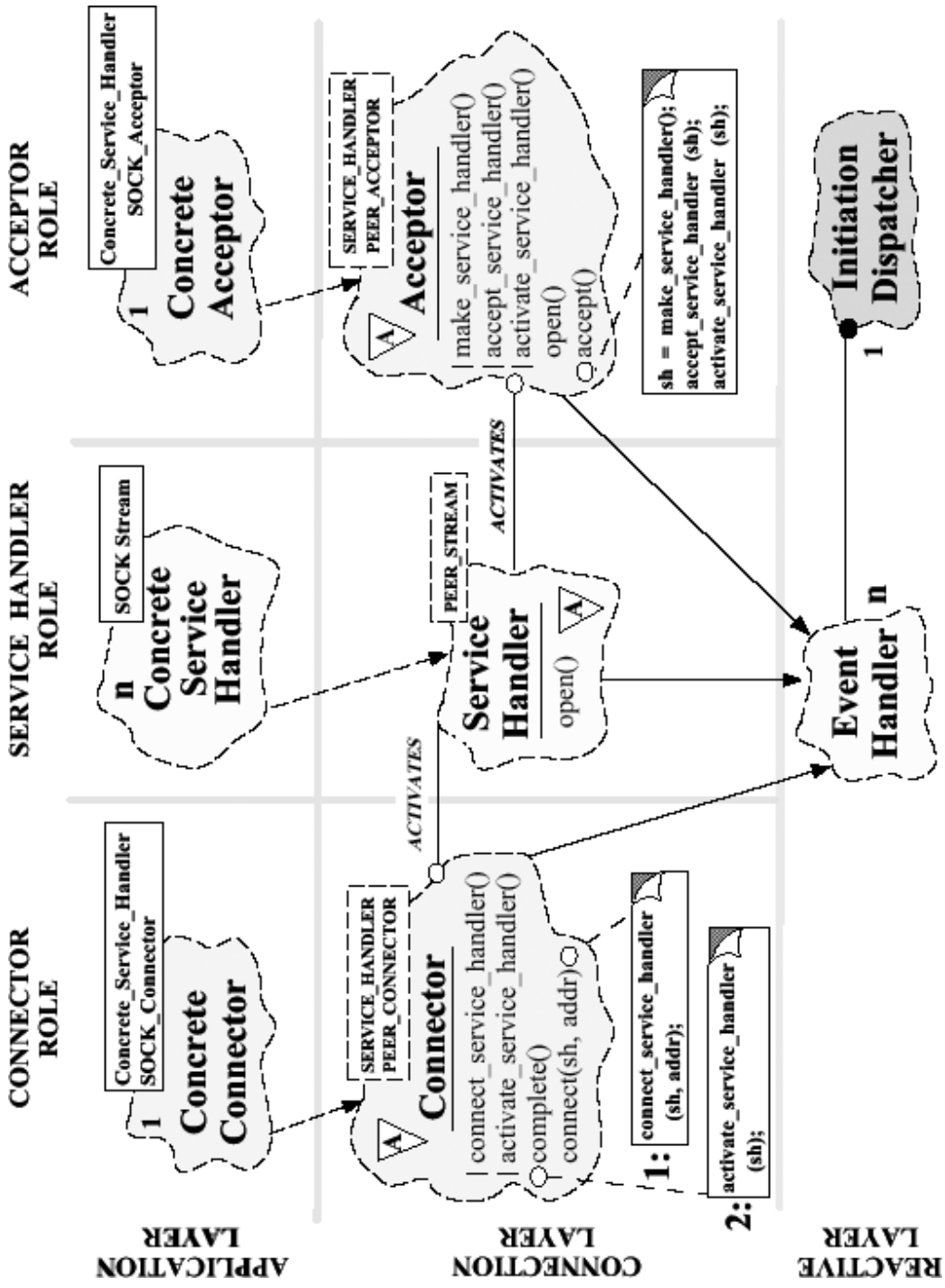

图9-6 Acceptor-Connector模式实现中的参与者的分层和划分

下面对接受器 - 连接器模式实现的讨论从底部的反应层开始，并向上通过连接层和应用层。

9.8.1 反应层

反应层处理发生在由I/O句柄表示的传输端点（比如socket端点）上的事件。该层的两个参与者，Initiation Dispatcher（发起分派器）和Event Handler（事件处理器），是由反应堆（Reactor）模式[3]定义的。该模式使得程序在单线程控制中就能够高效地完成来自多个来源的多种类型的事件的多路分离。

反应层中的两个主要角色是：

事件处理器：它规定由挂钩方法[7]组成的接口，抽象地表示应用可提供的事件处理操作。例如，这些挂钩方法表示这样一些事件：新连接请求、异步开始的连接请求的完成，或是来自相连对端的数据的到达，等等。Acceptor和Connector组件是从Event Handler派生的具体的事件处理器。

发起分派器：为登记、移除和分派Event Handler定义接口。Synchronous Event Demultiplexer（同步的事件多路分离器），比如select[8]或WaitForMultipleObjects[9]，通知Initiation Dispatcher何时回调应用特有的事件处理器，以响应特定类型的事件。常用事件包括连接接受事件、数据输入和输出事件，以及超时事件。

注意Initiation Dispatcher是9.6描述的Dispatcher的实现。一般而言，接受器 - 连接器Dispatcher可以是反应式、前摄式（Proactive）或多线程的。在这一实现中的特定的Initiation Dispatcher使用反应式模型来在单线程控制中多路分离和分派具体的事件处理器。在我们的例子中，Initiation Dispatcher是单体（Singleton）[10]，因为我们只需要它的一个实例用于整个进程。

9.8.2 连接层

连接层：

1. 创建Service Handler；
2. 被动地或主动地将Service Handler连接到它们的远地对端；以及
3. 一旦连接，启用Service Handler。

在此层中的所有行为都是完全通用的。特别地，注意下面描述的实现中的类是怎样委托具体的IPC机制和Concrete Service Handler的；后者是由在9.8.3中描述的应用层实例化的。

应用层委托连接层的方式与连接层委托反应层的方式相类似。例如，反应层中的Initiation Dispatcher代表连接层处理与初始化有关的事件，比如异步的建立连接。

在连接层中有三个主要角色：Service Handler（服务处理器）、Acceptor和Connector。

- **服务处理器：**该抽象类继承自Event_Handler，并为客户、服务器或同时扮演两种角色的组件所提供的服务处理提供通用接口。应用必须通过继承来定制此类，以执行特定类型的服务。Service Handler接口如下所示：

```
// PEER_STREAM is the type of the

// Concrete IPC mechanism.

template <class PEER_STREAM>
```

```
class Service_Handler : public Event_Handler
{
public:
// Pure virtual method (defined by a subclass).
virtual int open (void) = 0;

// Accessor method used by Acceptor and
// Connector to obtain the underlying stream.
PEER_STREAM &peer (void)
{
       return peer_stream_;
}

// Return the address that we're connected to.
PEER_STREAM::PEER_ADDR &remote_addr (void)
{
       return peer_stream_.remote_addr ();
}

protected:
// Concrete IPC mechanism instance.
PEER_STREAM peer_stream_;
};
```

一旦Acceptor或Connector建立了连接，它们调用Service Handler的open挂钩。该纯虚方法必须被Concrete Service Handler子类定义；后者执行服务特有的初始化和后续处理。

连接器：该抽象类实现主动连接建立和初始化Service Handler的通用策略。它的接口如下所示：

```
// The SERVICE_HANDLER is the type of service.
// The PEER_CONNECTOR is the type of concrete
// IPC active connection mechanism.
template <class SERVICE_HANDLER,
class PEER_CONNECTOR>
class Connector : public Event_Handler
```

```
{
public:
enum Connect_Mode
{
       SYNC, // Initiate connection synchronously.
       ASYNC // Initiate connection asynchronously.
};

// Initialization method.
Connector (void);

// Actively connecting and activate a service.
int connect (SERVICE_HANDLER *sh,
                           const PEER_CONNECTOR::PEER_ADDR &addr,
                           Connect_Mode mode);

protected:
// Defines the active connection strategy.
virtual int connect_service_handler(SERVICE_HANDLER *sh,
                           const PEER_CONNECTOR::PEER_ADDR &addr,
                           Connect_Mode mode);

// Register the SERVICE_HANDLER so that it can
// be activated when the connection completes.
int register_handler (SERVICE_HANDLER *sh, Connect_Mode mode);

// Defines the handler's concurrency strategy.
virtual int activate_service_handler(SERVICE_HANDLER *sh);

// Activate a SERVICE_HANDLER whose
// non-blocking connection completed.
virtual int complete (HANDLE handle);

private:
// IPC mechanism that establishes
// connections actively.
```

```
PEER_CONNECTOR connector_;


// Collection that maps HANDLEs

// to SERVICE_HANDLER *s.

Map_Manager<HANDLE, SERVICE_HANDLER *>handler_map_;


// Inherited from the Event_Handler -- will be

// called back by Eactor when events complete

// asynchronously.

virtual int handle_event (HANDLE, EVENT_TYPE);

};


// Useful "short-hand" macros used below.

#define SH SERVICE_HANDLER

#define PC PEER_CONNECTOR
```

Conncetor通过特定类型的PEER CONNECTOR和SERVICE HANDLER被参数化。PEER CONNECTOR提供的传输机制被Connector用于主动地建立连接，或是同步地、或是异步地。SERVICE HANDLER提供的服务对与相连的对端交换的数据进行处理。C++参数化类型被用于使（1）连接建立策略与（2）服务处理器类型、网络编程接口和传输层连接协议去耦合。

参数化类型是有助于提高可移植性的实现决策。例如，它们允许整体地替换Connector所用的IPC机制。这使得Connector的连接建立代码可在含有不同网络编程接口（例如，有socket，但没有TLI；反之亦然）的平台间进行移植。例如，取决于平台是支持socket还是TLI[11]，PEER CONNECTOR模板参数可以通过SOCK Connector或TLI Connector来实例化。使用参数化类型的另一个动机是改善运行时效率，因为模板实例化发生在编译时。

更为动态的去耦合可以经由继承和多态、通过使用[10]中描述的工厂方法（Factory Method）和策略（Strategy）模式来完成。例如，Connector可以存储指向PEER CONNECTOR基类的指针。根据从工厂返回的PEER CONNECTOR的子类，这个PEER CONNECTOR的connect方法可在运行时被动态地绑定。一般而言，在参数化类型和动态绑定之间的权衡是参数化类型可能带来额外的编译/链接时开销，而动态绑定可能带来额外的运行时开销。

connect方法是应用用以通过Connector来发起连接的入口。它的实现如下所示：

```
template <class SH, class PC> int

Connector<SH, PC>::connect(SERVICE_HANDLER *service_handler,

const PEER_CONNECTOR::PEER_ADDR &addr,

Connect_Mode mode)

{

connect_service_handler (service_handler, addr, mode);
```

```
}
```

该方法使用桥接（Bridge）模式[10]来使Concrete Connector能透明地修改连接策略，而不用改变组件接口。为此，connect方法委托Connector的连接策略，connect_service_handler，来发起连接。如下所示：

```
template <class SH, class PC> int

Connector<SH, PC>::connect_service_handler

(SERVICE_HANDLER *service_handler,

const PEER_CONNECTOR::PEER_ADDR &remote_addr,

Connect_Mode mode)

{

// Delegate to concrete PEER_CONNECTOR

// to establish the connection.

if (connector_.connect (*service_handler,

                                                remote_addr,

                                                mode)
                                                == -1)

{

       if (mode == ASYNC && errno == EWOULDBLOCK)

       {

              // If connection doesn't complete immediately

              // and we are using non-blocking semantics

              // then register this object with the

              // Initiation_Dispatcher Singleton so it will

              // callback when the connection is complete.

              Initiation_Dispatcher::instance

                            ()->register_handler (this, WRITE_MASK);

              // Store the SERVICE_HANDLER in the map of

              // pending connections.

              handler_map_.bind

                            (connector_.get_handle (), service_handler);

              }

}

else if (mode == SYNC)

       // Activate if we connect synchronously.

       activate_service_handler (service_handler);
```

}

如图9-7所示，如果Connect_Mode参数的值是SYNC，一旦连接同步地完成，SERVICE HANDLER将会被启用。该图与图9-4相类似，但是提供了另外的实现细节，比如get_handle和handle_event挂钩方法的使用。

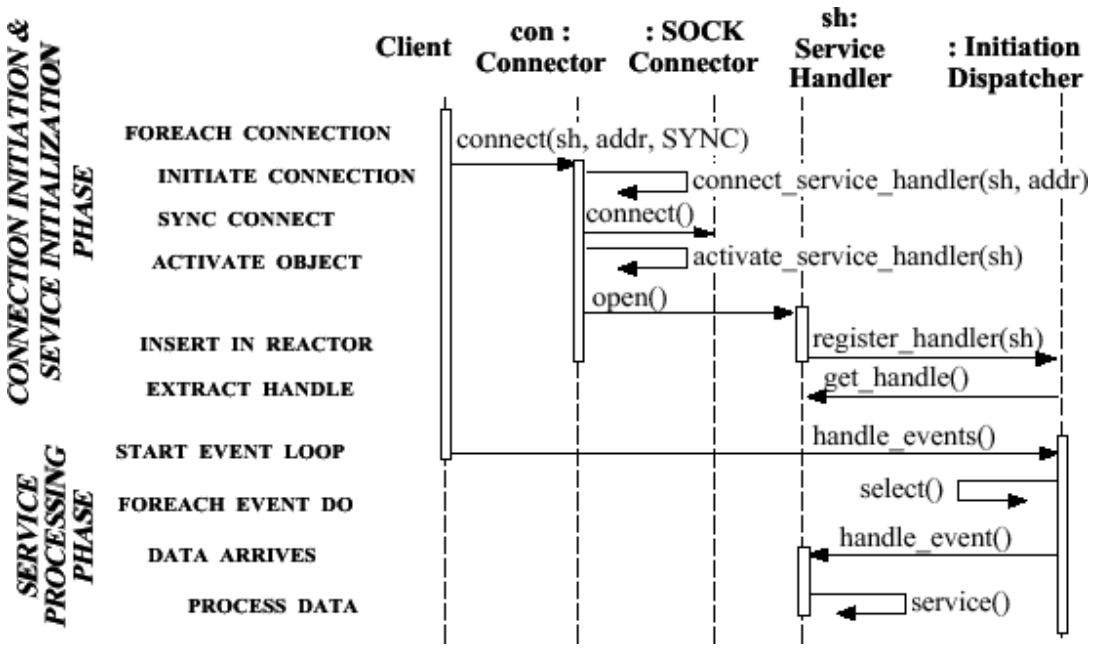

图9-7 用于同步连接的Connector参与者之间的协作

为高效地与多个Peer相连，Connector可能还需要主动、异步地建立连接，也就是，不阻塞调用者。如图9-8所示，异步行为通过将ASYNC连接模式传递给Connector::connect来指定。该图与图9-5相类似，但是还提供了其他与当前实现相应的细节。

一旦实例化，PEER CONNECTOR类提供具体的IPC机制来同步或异步地发起连接。这里所显示的Connector模式的实现使用OS和通信协议栈所提供的异步连接机制。例如，在UNIX或Win32上，Connector可以将socket设置进非阻塞模式，并使用像select或WaitForMultipleObject这样的事件多路分离器来确定连接何时完成。

为处理还未完成的异步连接，Connector维护Service Handler映射表。因为Connector继承自Event Handler，当连接完成时，Initiation Dispatcher可以自动回调Connector的handle_event方法。

handle_event方法是一个适配器（Adapter）[10]，它将Initiation Dispatcher的事件处理接口转换为对Connector模式的complete方法的调用。

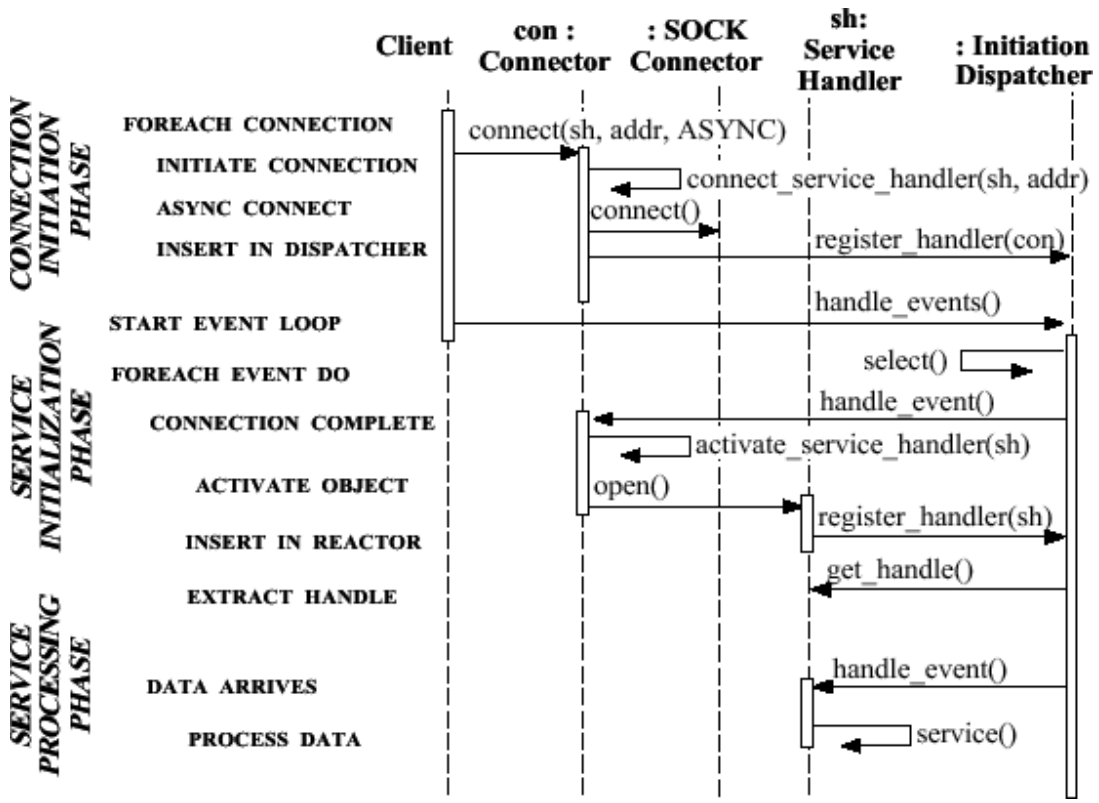

图9-8 用于异步连接的Connector参与者之间的协作

Connector的handle_event方法如下所示：

```
template <class SH, class PC> int
Connector<SH, PC>::handle_event (HANDLE handle, EVENT_TYPE type)
{
// Adapt the Initiation_Dispatcher's event
// handling API to the Connector's API.
complete (handle);
}
```

complete方法启用刚刚成功完成非阻塞连接的SERVICE HANDLER，如下所示：

```
template <class SH, class PC> int
Connector<SH, PC>::complete (HANDLE handle)
{
SERVICE_HANDLER *service_handler = 0;
```

```
// Locate the SERVICE_HANDLER corresponding

// to the HANDLE.

handler_map_.find (handle, service_handler);


// Transfer I/O handle to SERVICE_HANDLER *.

service_handler->set_handle (handle);


// Remove handle from Initiation_Dispatcher.

Initiation_Dispatcher::instance

       ()->remove_handler (handle, WRITE_MASK);


// Remove handle from the map.

handler_map_.unbind (handle);


// Connection is complete, so activate handler.

activate_service_handler (service_handler);

}
```

complete方法在其内部映射表中查找并移除已连接的SERVICE HANDLER，并将I/O句柄传递给SERVICE HANDLER。最后，它通过调用activate_service_handler方法初始化SERVICE HANDLER。该方法委托由SERVICE HANDLER的open挂钩指定的并发策略。如下所示：

```
template <class SH, class PC> int

Connector<SH, PC>::activate_service_handler

(SERVICE_HANDLER *service_handler)

{

service_handler->open ();

}
```

Service Handler的open挂钩在连接成功建立时被调用。注意该挂钩都将被调用，不管（1）连接是同步还是异步发起的，或（2）它们是被主动还是被动连接的。这样的统一性使得开发者有可能编写这样的Service Handler，其处理可以完全地与它们是怎样被连接和初始化的去耦合。

接受器（Acceptor）：该抽象类为被动连接建立和初始化Service Handler实现通用的策略。Acceptor的接口如下所示：

```
// The SERVICE_HANDLER is the type of service.
// The PEER_ACCEPTOR is the type of concrete
// IPC passive connection mechanism.
template <class SERVICE_HANDLER,
       class PEER_ACCEPTOR>
class Acceptor : public Event_Handler
{
public:
// Initialize local_addr transport endpoint factory
// and register with Initiation_Dispatcher Singleton.
virtual int open(const PEER_ACCEPTOR::PEER_ADDR &local_addr);

// Factory Method that creates, connects, and
// activates SERVICE_HANDLER's.
virtual int accept (void);

protected:
// Defines the handler's creation strategy.
virtual SERVICE_HANDLER *make_service_handler (void);

// Defines the handler's connection strategy.
virtual int accept_service_handler(SERVICE_HANDLER *);

// Defines the handler's concurrency strategy.
virtual int activate_service_handler(SERVICE_HANDLER *);

// Demultiplexing hooks inherited from Event_Handler,
// which is used by Initiation_Dispatcher for
// callbacks.
virtual HANDLE get_handle (void) const;
virtual int handle_close (void);

// Invoked when connection requests arrive.
virtual int handle_event (HANDLE, EVENT_TYPE);

private:
```

```
// IPC mechanism that establishes

// connections passively.

PEER_ACCEPTOR peer_acceptor_;

};


// Useful "short-hand" macros used below.

#define SH SERVICE_HANDLER

#define PA PEER_ACCEPTOR
```

Acceptor通过特定类型的PEER ACCEPTOR和SERVICE HANDLER被参数化。PEER ACCEPTOR提供的传输机制被Acceptor用于被动地建立连接。SERVICE HANDLER提供的服务对与远地对端交换的数据进行处理。注意SERVICE HANDLER是由应用层提供的具体的服务处理器。

参数化类型使Acceptor的连接建立策略与服务处理器的类型、网络编程接口及传输层连接发起协议去耦合。就如同Connector一样，通过允许整体地替换Acceptor所用的机制，参数化类型的使用有助于提高可移植性。这使得连接建立代码可在含有不同网络编程接口（比如有socket，但没有TLI；反之亦然）的平台间移植。例如，取决于平台能够更为高效地支持socket还是TLI，PEER ACCEPTOR模板参数可以通过SOCK Acceptor或TLI Acceptor来实例化。

下面给出Acceptor的方法的实现。应用通过调用Acceptor的open方法来将其初始化。如下所示：

```
template <class SH, class PA> int

Acceptor<SH, PA>::open

(const PEER_ACCEPTOR::PEER_ADDR &local_addr)

{

// Forward initialization to the PEER_ACCEPTOR.

peer_acceptor_.open (local_addr);


// Register with Initiation_Dispatcher, which

// ‘‘double-dispatches’’ without get_handle()

// method to extract the HANDLE.

Initiation_Dispatcher::instance

       ()->register_handler (this, READ_MASK);

}
```

local_addr被传递给open方法。该参数含有网络地址，例如，本地主机的IP地址和TCP端口号，用于侦听连接。Open方法将此地址转发给PEER ACCEPTOR定义的被动连接接受机制。该机制初始化传输端点工厂，由后者将地址广告给有兴趣与此Acceptor连接的客户。

传输端点工厂的行为由用户所实例化的PEER ACCEPTOR的类型来决定。例如，它可以是socket[13]、TLI[14]、STREAM管道[15]、Win32命名管道等的C++包装。

在传输端点工厂被初始化后，open方法将其自身登记到Initiation Dispatcher。Initiation Dispatcher执行“双重分派”，回调Acceptor的get_handle方法，以获取底层传输端点工厂的HANDLE。如下所示：

```
template <class SH, class PA> HANDLE

Acceptor<SH, PA>::get_handle (void)

{

return peer_acceptor_.get_handle ();

}
```

Initiation Dispatcher在内部表中存储此HANDLE。Synchronous Event Demultipler（同步事件多路分离器），比如select，随即被用于检测和多路分离到来的来自客户的连接请求。因为Acceptor类继承自Event Handler，当连接从对端到达时，Initiation Dispatcher可以自动回调Acceptor的handle_event方法。该方法是一个适配器（Adapter），它将Initiation Dispatcher的事件处理接口转换为对Acceptor的accept方法的调用。如下所示：

```
template <class SH, class PA> int

Acceptor<SH, PA>::handle_event (HANDLE, EVENT_TYPE)

{

// Adapt the Initiation_Dispatcher’s event handling

// API to the Acceptor’s API.

accept ();

}
```

如下所示，accept方法是一个模板方法（Template Method）[10]，它为创建新SERVICE HANDLER、将连接接受进其中并启用服务而实现接受器－连接器模式的被动初始化策略：

```
template <class SH, class PA> int

Acceptor<SH, PA>::accept (void)

{

// Create a new SERVICE_HANDLER.

SH *service_handler = make_service_handler ();
```

```
// Accept connection from client.
accept_service_handler (service_handler);

// Activate SERVICE_HANDLER by calling
// its open() hook.
activate_service_handler (service_handler);
}
```

该方法非常简洁，因为它将所有低级细节都分解进具体的SERVICE HANDLER和PEER ACCEPTOR中，后二者通过参数化类型被实例化，并可被Acceptor的子类定制。特别地，因为accept是模板方法，子类可以扩展Acceptor的任意或所有的连接建立和初始化策略。这样的灵活性使得开发者有可能编写这样的Service Handler，其行为与它们被被动地连接和初始化的方式是相分离的。

make_service_handler工厂方法定义Acceptor用于创建SERVICE HANDLER的缺省策略。如下所示：

```
template <class SH, class PA> SH *
Acceptor<SH, PA>::make_service_handler (void)
{
return new SH;
}
```

缺省行为使用了“请求策略”（demand strategy），它为每个新连接创建新的SERVICE HANDLER。但是，Acceptor的子类可以重定义这一策略，以使用其他策略创建SERVICE HANDLE，比如创建单独的单体（Singleton）[10]或从共享库中动态链接SERVICE HANDLER。

accept_service_handler方法在下面定义Acceptor所用的SERVICE HANDLER连接接受策略：

```
template <class SH, class PA> int
Acceptor<SH, PA>::accept_service_handler(SH *handler)
{
peer_acceptor_->accept (handler->peer ());
}
```

缺省行为委托PEER ACCEPTOR所提供的accept方法。子类可以重定义accept_service_handler方法，以执行更为复杂的行为，比如验证客户的身份，以决定是接受还是拒绝连接。

Activate_service_handler定义Acceptor的SERVICE HANDLER并发策略：

```
template <class SH, class PA> int

Acceptor<SH, PA>::activate_service_handler(SH *handler)

{

handler->open ();

}
```

该方法的缺省行为是通过调用SERVICE HANDLER的open挂钩将其启用。这允许SERVICE HANDLER选择它自己的并发策略。例如，如果SERVICE HANDLER继承自Event Handler，它可以登记到Initiation Dispatcher，从而在事件发生在SERVICE HANDLER的PEER STREAM通信端点上时，使Initiation Dispatcher能够分派其handle_event方法。Concrete Acceptor可以重定义此策略，以完成更为复杂的并发启用。例如，子类可以使SERVICE HANDLER成为主动对象（Active Object）[5]，使用多线程或多进程来处理数据。

当Acceptor终止时，无论是由于错误还是由于整个应用的关闭，Initiation Dispatcher都调用Acceptor的handle_close方法，后者可以释放任何动态获取的资源。在此例中，handle_close方法简单地将close请求转发给PEER ACCEPTOR的传输端点工厂。如下所示：

```
template <class SH, class PA> int

Acceptor<SH, PA>::handle_close (void)

{

peer_acceptor_.close ();

}
```

9.8.3 应用层

应用层提供具体的进程间通信（IPC）机制和具体的Service Handler。IPC机制被封装在C++类中，以简化编程、增强复用，并使开发者能够整个地替换IPC机制。例如，9.9中使用的SOCK Acceptor、SOCK Connector，以及SOCK Stream类是ACE C++ socket包装类库[11]的一部分。它们通过高效、可移植和类型安全的C++包装来封装像TCP和SPX这样的面向连接协议的面向流的语义。

应用层中的三个主要角色描述如下：

具体的服务处理器（Concrete Service Handler）：该类定义具体的应用服务，由Concrete Acceptor或Concrete Connector启用。Concrete Service Handler通过特定类型的C++ IPC包装（它与和其相连的对端进行数据交换）来实例化。

具体的连接器（Concrete Connector）：该类通过具体的参数化类型参数SERVICE HANDLER和PEER CONNECTOR来实例化通用的Connector工厂。

具体的接受器（Concrete Acceptor）：该类通过具体的参数化类型参数SERVICE HANDLER和PEER ACCEPTOR来实例化通用的Acceptor工厂。

Concrete Service Handler还可以定义服务的并发策略。例如，Service Handler可以从Event Handler继承，并采用反应堆（Reactor）[3]模式来在单线程控制中处理来自对端的数据。相反，Service Handler也可以使用主动对象（Active Object）模式[5]处理到来的数据，而其所在线程控制与Acceptor连接它所用的不相同。下面，我们为我们的Gateway例子实现Concrete Service Handler，演示怎样灵活地配置若干不同的并发策略，而又不影响接受器－连接器模式的结构或行为。

在9.9的示例代码中，SOCK Connector和SOCK Acceptor是分别用于主动和被动地建立连接的IPC机制。同样地，SOCK Stream被用作数据传输递送机制。但是，通过其他机制（比如TLI Connector或Named Pipe Acceptor）来参数化Connector和Acceptor也是相当直接的，因为IPC机制被封装在C++包装类中。同样地，通过使用不同的PEER STREAM，（比如SVR4 UNIX TLI Stream或Win32 Named Pipe Stream）来参数化Concrete Service Handler，很容易改变数据传输机制。

9.9演示怎样实例化Concrete Service Handler、Concrete Connector和Concrete Acceptor，实现9.2中描述的Peer和Gateway。这个特定的应用层例子定制连接层中的Connector和Acceptor组件所提供的通用初始化策略。

9.9 例子解答

下面的代码演示9.2中描述的Peer和Gateway怎样使用接受器－连接器模式来简化连接建立和服务初始化。9.9.1演示Peer怎样扮演被动角色，9.9.2演示Gateway怎样在与被动的Peer的连接建立中扮演主动角色。

9.9.1 用于对端的具体组件

图9-9演示Concrete Acceptor和Concrete Service Handler组件是怎样在Peer中构造的。该图中的Acceptor组件与图9-11中的Connector组件是互补的。

用于与Gateway通信的服务处理器：如下所示的Status Handler、Bulk Data Handler和Command Handler类处理发送到Gateway和从Gateway接收的路由消息。因为这些Concrete Service Handler类继承自Service Handler，它们可以被Acceptor被动地初始化。

为演示接受器－连接器模式的灵活性，这些Service Handler中的每个open例程都可以实现不同的并发策略。特别地，当Status Handler被启用时，它运行在单独的线程中；Bulk Data Handler作为单独的进程运行；而Command Handler运行在与Initiation Dispatcher相同的线程中，后者为Acceptor工厂进行连接请求的多路分离。注意这些并发策略的改变并不影响Acceptor的实现，它是通用的，因而也是高度灵活和可复用的。

我们从定义一个Service Handler开始，它为基于socket的数据传输使用SOCK Stream：

```
typedef Service_Handler <SOCK_Stream>PEER_HANDLER;
```

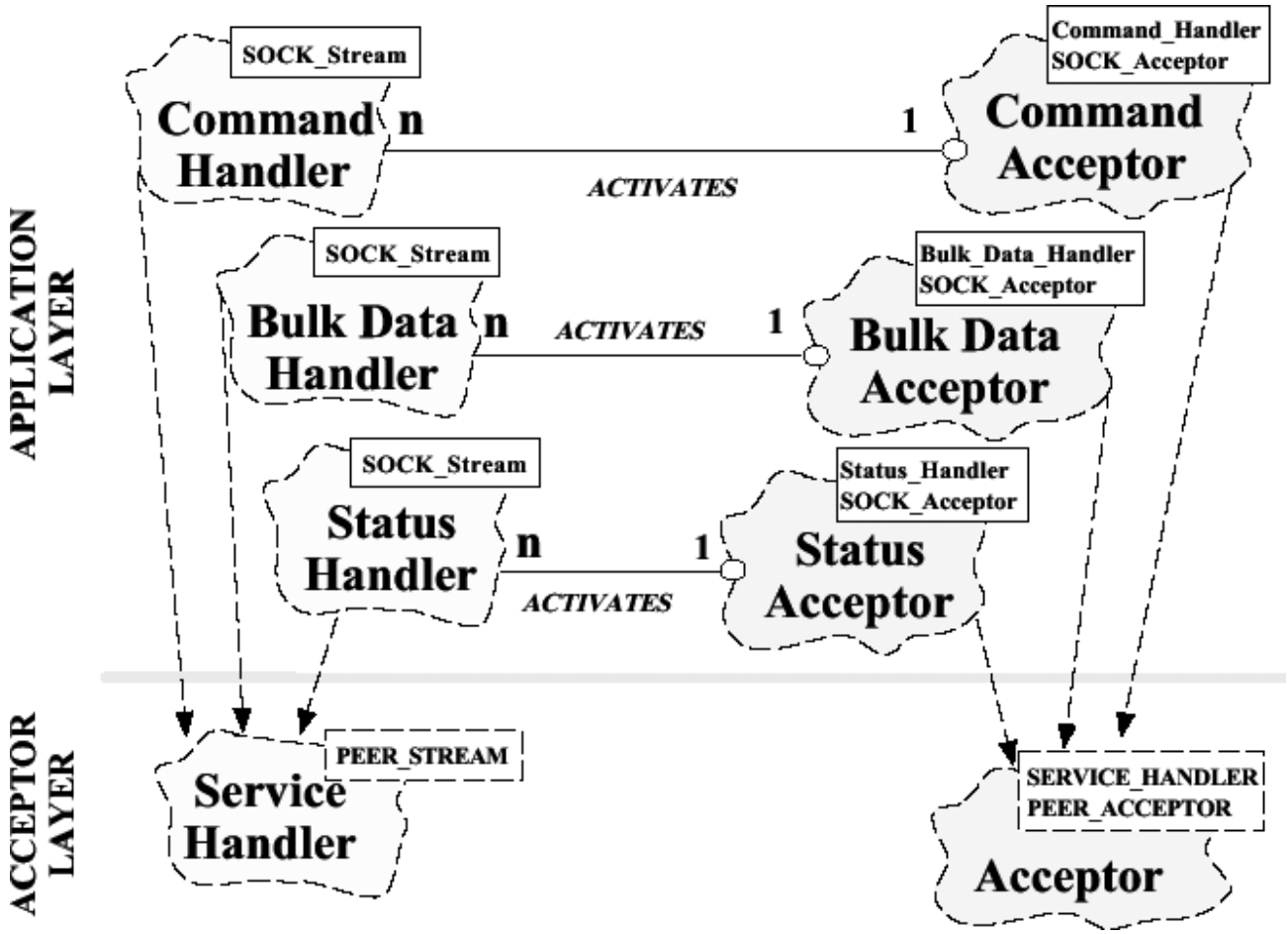

图9-9 对端的Acceptor参与者的结构

PEER HANDLER的typedef构成所有后续服务处理器的基础。例如，Status Handler类处理发送到Gateway和从Gateway接收的状态数据：

```
class Status_Handler : public PEER_HANDLER

{

public:

// Performs handler activation.

virtual int open (void)

{

       // Make handler run in separate thread (note

       // that Thread::spawn requires a pointer to

       // a static method as the thread entry point).

       Thread::spawn (&Status_Handler::service_run, this);

}
```

```
// Static entry point into thread, which blocks
// on the handle_event () call in its own thread.
static void *service_run (Status_Handler *this_)
{
       // This method can block since it
       // runs in its own thread.
       while (this_->handle_event () != -1)
              continue;
}

// Receive and process status data from Gateway.
virtual int handle_event (void)
{
       char buf[MAX_STATUS_DATA];
       stream_.recv (buf, sizeof buf);
       // ...
}

// ...
};
```

PEER HANDLER还可被子类化，以生成具体的服务处理器，处理大块数据和命令。例如，Bulk Data Handler类处理发送到Gateway和从Gateway接收的大块数据：

```
class Bulk_Data_Handler : public PEER_HANDLER
{
public:
// Performs handler activation.
virtual int open (void)
{
       // Handler runs in separate process.
       if (fork () == 0) // In child process.

       // This method can block since it
       // runs in its own process.
       while (handle_event () != -1)
```

```
            continue;

        // ...
}

// Receive and process bulk data from Gateway.
virtual int handle_event (void)
{
        char buf[MAX_BULK_DATA];
        stream_.recv (buf, sizeof buf);
        // ...
}

// ...
};
```

Command Handler类处理发送到Gateway和从Gateway接收的命令。

```
class Command_Handler : public PEER_HANDLER
{
public:
// Performs handler activation.
virtual int open (void)
{
        // Handler runs in same thread as main
        // Initiation_Dispatcher singleton.
        Initiation_Dispatcher::instance
                ()->register_handler (this, READ_MASK);
}

// Receive and process command data from Gateway.
virtual int handle_event (void)
{
        char buf[MAX_COMMAND_DATA];
```

```
        // This method cannot block since it borrows

        // the thread of control from the

        // Initiation_Dispatcher.

        stream_.recv (buf, sizeof buf);


        // ...

}


//...

};
```

用于创建Peer Service Handler**的接受器：**如下所示的s_acceptor、bd_acceptor和c_acceptor对象是Concrete Acceptor**工厂实例，它们分别创建并启用**Status Handler、Bulk Data Handler**和**Command Handler**。

```
// Accept connection requests from Gateway and

// activate Status_Handler.

Acceptor<Status_Handler, SOCK_Acceptor> s_acceptor;


// Accept connection requests from Gateway and

// activate Bulk_Data_Handler.

Acceptor<Bulk_Data_Handler, SOCK_Acceptor> bd_acceptor;


// Accept connection requests from Gateway and

// activate Command_Handler.

Acceptor<Command_Handler, SOCK_Acceptor> c_acceptor;
```

注意模板和动态绑定的使用是怎样允许特定细节灵活地变化的。特别地，在整个这一部分中，当并发策略被修改时，没有 Acceptor**组件发生变化。这样的灵活性的原因是并发策略已被分解进**Service Handler**中，而不是与**Acceptor**耦合在一起。**

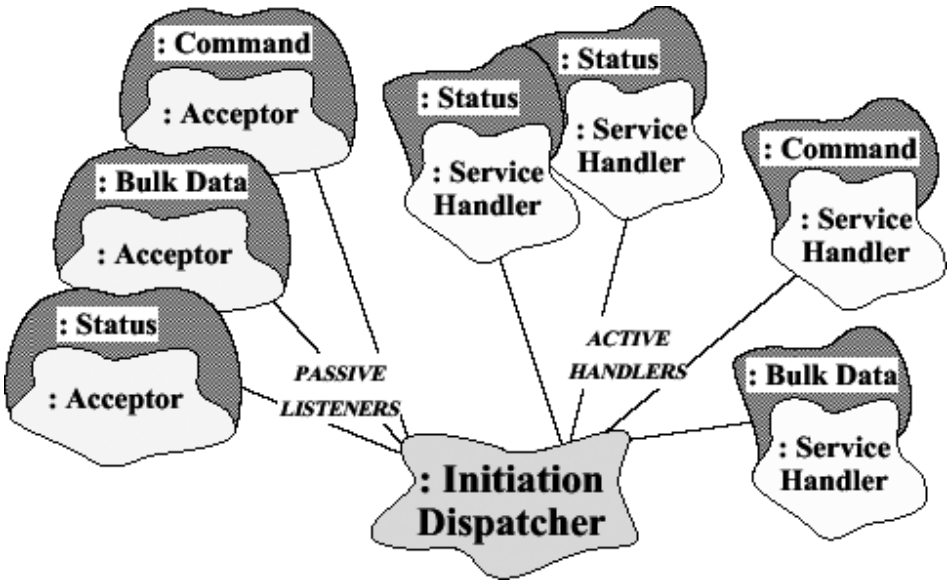

图9-10 对端中的Acceptor组件的对象图

Peer**主函数：**主程序通过调用具体的Acceptor工厂的open挂钩（以每个服务的TCP端口为参数）来对它们进行初始化。如9.8.2所示，每个Acceptor工厂自动地在它的open方法中将其自身登记到Initiation Dispatcher的实例。

```
// Main program for the Peer.
int main (void)
{
// Initialize acceptors with their
// well-known ports.
s_acceptor.open (INET_Addr (STATUS_PORT));
bd_acceptor.open (INET_Addr (BULK_DATA_PORT));
c_acceptor.open (INET_Addr (COMMAND_PORT));

// Event loop that handles connection request
// events and processes data from the Gateway.
for (;;)
       Initiation_Dispatcher::instance()->handle_events ();
}
```

一旦Acceptor被初始化，主程序进入事件循环，使用Initiation Dispatcher来检测来自Gateway的连接请求。当连接到达时，Initiation Dispatcher回调适当的Acceptor，由其创建适当的PEER HANDLER来执行服务、将连接接受进处理器、并启用处理器。

图9-10演示在与Gateway（如图9-12所示）的四个连接被建立、以及四个Service Handler被创建和启用后，Peer中的Concrete Acceptor组件之间的关系。在Concrete Service Handler与Gateway交换数据的同时，三个Acceptor也在主线程中持续地侦听新连接。

9.9.2 用于Gateway的具体组件

图9-11演示Concrete Connector和Concrete Service Handler组件是怎样在假想的Gateway配置中构造的。该图中的Connector组件与图9-9中的Acceptor组件是互补的。

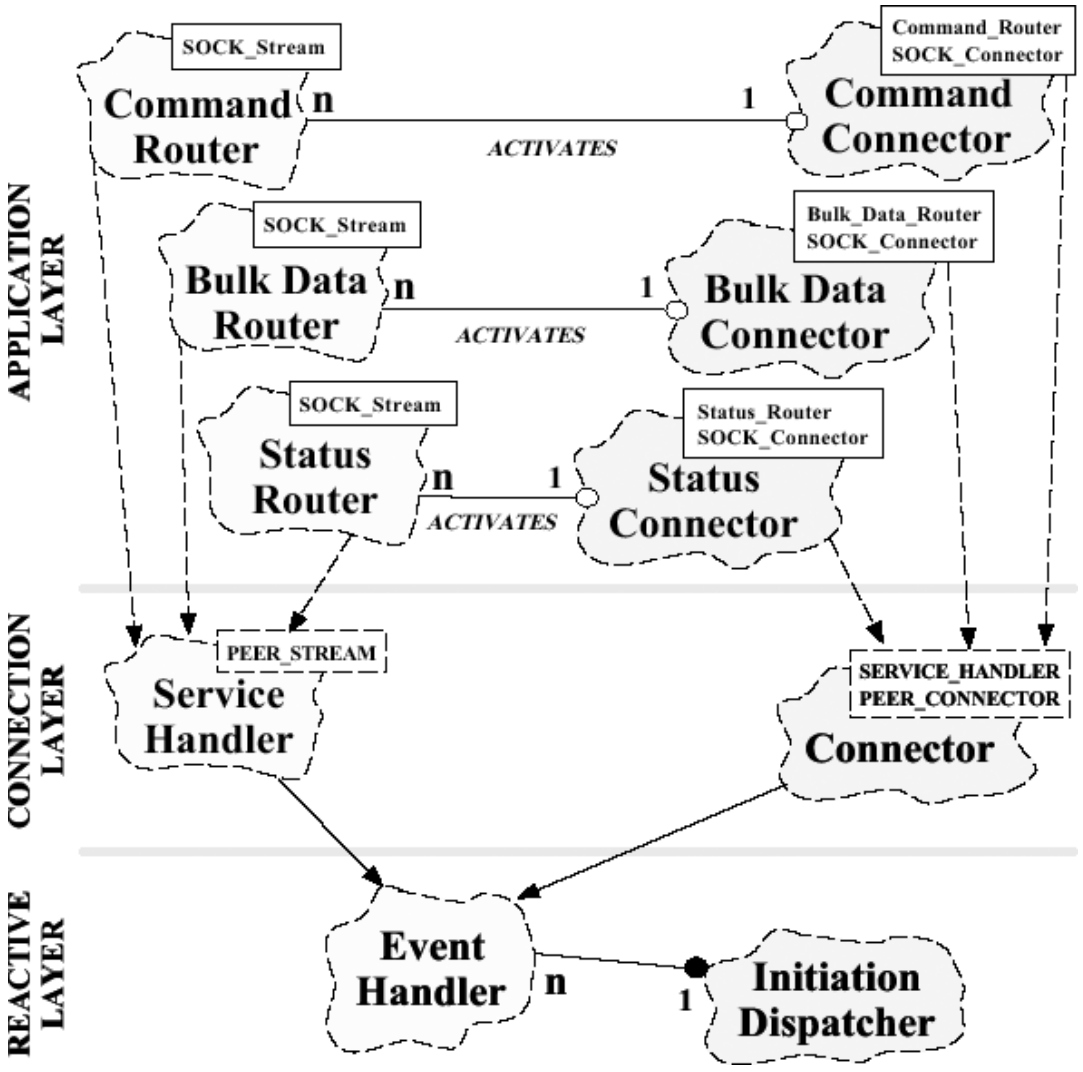

图9-11 网关的Connector参与者的结构

用于Gateway路由的服务处理器：如下所示的Status Router、Buld Data Router和Command Router类将它们从源Peer接收到的数据路由到一或多个目的Peer。因为这些Concrete Service Handler类继承自Service Handler，它们可以被Connector主动地连接和初始化。

为演示接受器 - 连接器模式的灵活性，Service Handler中的每个open例程实现不同的并发策略。特别地，当Status Router被启用时，它运行在单独的线程中；Bulk Data Router作为单独的进程运

行；而Command Router运行在与Initiation Dispatcher相同的线程中，后者为Connector工厂进行连接完成事件的多路分离。就如同Acceptor一样，注意这些并发策略的变动并不会影响Connector的实现，它是通用的，因而也是高度灵活和可复用的。

我们从定义一个Service Handler开始，它为基于socket的数据传输而定制：

```
typedef Service_Handler <SOCK_Stream>PEER_ROUTER;
```

该类构成所有后续路由服务的基础。例如，Status Router类路由状态数据到Peer，或路由来自Peer的数据：

```
class Status_Router : public PEER_ROUTER
{
public:
// Activate router in separate thread.
virtual int open (void)
{
       // Thread::spawn requires a pointer to a
       // static method as the thread entry point).
       Thread::spawn (&Status_Router::service_run, this);
}

// Static entry point into thread, which blocks
// on the handle_event() call in its own thread.
static void *service_run (Status_Router *this_)
{
       // This method can block since it
       // runs in its own thread.
       while (this_->handle_event () != -1)
              continue;
}

// Receive and route status data from/to Peers.
virtual int handle_event (void)
{
       char buf[MAX_STATUS_DATA];
       peer_stream_.recv (buf, sizeof buf);
```

```
        // Routing takes place here...
}

// ...
};
```

PEER ROUTER可被子类化，以生成用于路由大块数据和命令具体的服务处理器。例如，Bulk Data Router路由数据到Peer，或路由来自Peer的数据：

```
class Bulk_Data_Router : public PEER_ROUTER
{
public:
// Activates router in separate process.
virtual int open (void)
{
        if (fork () == 0) // In child process.
                // This method can block since it
                // runs in its own process.
                while (handle_event () != -1)
                        continue;

                // ...
}

// Receive and route bulk data from/to Peers.
virtual int handle_event (void)
{
        char buf[MAX_BULK_DATA];
        peer_stream_.recv (buf, sizeof buf);

        // Routing takes place here...
}
};
```

Command Router类路由命令数据到Peer，或路由来自Peer的命令数据：

```
class Command_Router : public PEER_ROUTER

{

public:

// Activates router in same thread as Connector.

virtual int open (void)

{

       Initiation_Dispatcher::instance

              ()->register_handler (this, READ_MASK);

}


// Receive and route command data from/to Peers.

virtual int handle_event (void)

{

       char buf[MAX_COMMAND_DATA];


       // This method cannot block since it borrows the

       // thread of control from the Initiation_Dispatcher.

       peer_stream_.recv (buf, sizeof buf);


       // Routing takes place here...

}

};
```

用于创建Peer Service Handler**的接受器：**下面的typedef定义为PEER ROUTER而定制的connector工厂：

```
typedef Connector<PEER_ROUTERS, SOCK_Connector>PEER_CONNECTOR;
```

不像 Concrete Acceptor组件，我们只需要单个Concrete Connector。原因是每个Concrete Acceptor都被用作创建特定类型的Concrete Service Handler（比如Bulk Data Handler或Command Handler）的工厂。因此，必须预先知道全部的类型，从而使多种Concrete Acceptor类型成为必要。相反，传递给Connector的connect方法的Concrete Service Handler是在外部初始化的。因此，它们可以统一地被当作PEER ROUTER处理。

Gateway主函数：Gateway的主程序如下所示。get_peer_addrs函数创建Status、Bulk Data和Command Router，通过Gateway路由消息。该函数（它的实现没有给出）从配置文件或名字服务中读取Peer地址列表。每个Peer地址含有IP地址和端口号。一旦Router被初始化，上面定义的Connector工厂通过将ASYNC标志传递给connect方法来异步地发起所有连接。

```
// Main program for the Gateway.

// Obtain an STL vector of Status_Routers,
// Bulk_Data_Routers, and Command_Routers
// from a config file.

void get_peer_addrs (vector<PEER_ROUTERS> &peers);

int main (void)
{
// Connection factory for PEER_ROUTERS.
PEER_CONNECTOR peer_connector;

// A vector of PEER_ROUTERs that perform
// the Gateway's routing services.
vector<PEER_ROUTER> peers;

// Get vector of Peers to connect with.
get_peer_addrs (peers);

// Iterate through all the Routers and
// initiate connections asynchronously.
for (vector<PEER_ROUTER>::iterator i = peers.begin ();
       i != peers.end ();
       i++)
{
       PEER_ROUTER &peer = *i;
       peer_connector.connect (peer,
                                              peer.remote_addr
                                              (),
                                              PEER_CONNECTOR::ASYNC);
}
```

```
// Loop forever handling connection completion

// events and routing data from Peers.

for (;;)

       Initiation_Dispatcher::instance()->handle_events ();


/* NOTREACHED */

}
```

所有连接都被异步地调用。它们通过Connector的complete方法并发地完成，该方法在Initiation Dispatcher的事件循环中被回调。此事件循环还为Command Router对象多路分离和分派路由事件；该对象运行在Initiation Dispatcher的线程控制中。Status Router和Bulk Data Router分别执行在单独的线程和进程中。

图9-12演示在与Peer（如图9-10所示）的四个连接被建立、以及四个Concrete Service Handler被创建和启用后，Gateway中的组件之间的关系。该图演示到另一个Peer的四个连接，它们被Connector“拥有”，还没有完成。当所有Peer连接完全建立时，Gateway将路由并转发由Peer发送给它的消息。

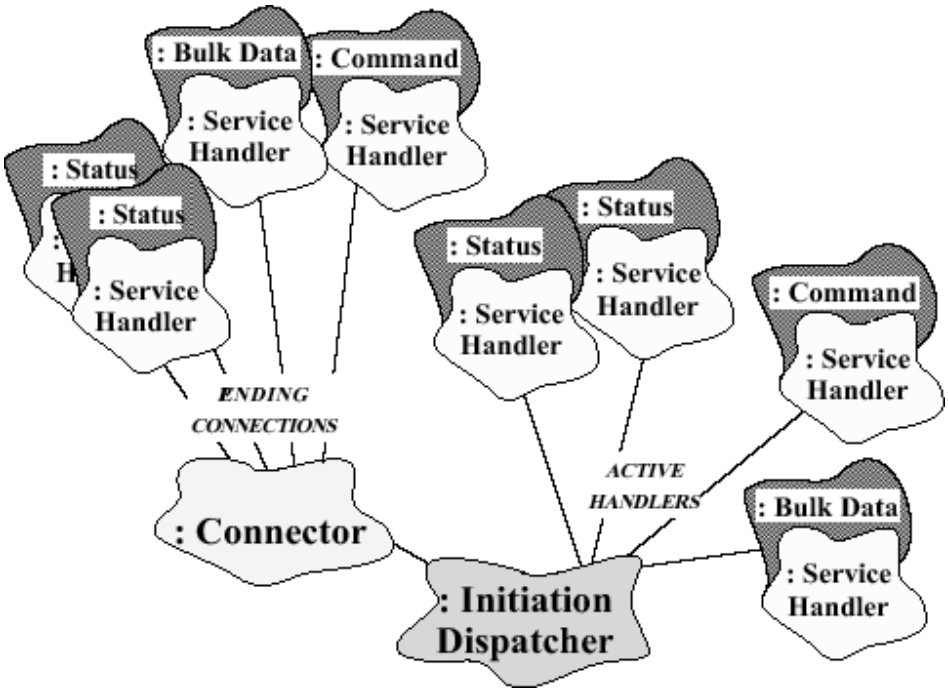

图9-12 网关中的Connector组件的对象图

9.10 已知应用

接受器 - 连接器模式已被用于广泛的构架、工具包和系统中：

UNIX**网络超级服务器：**比如inetd[13]、listen[14]以及来自ASX构架[6]的Service Configurator（服务配置器）看守。这些超级服务器利用主Acceptor进程，在一组通信端口上侦听连接。每个端口都和与通信有关的服务（比如标准的Internet服务ftp、telnet、daytime和echo）相关联。接受器进程使inetd超级服务器的功能分解为两个分离的部分：一个用于建立连接，另一个用于接收和处理来自对端的的请求。当服务请求在被监控的端口上到达时，接受器进程接受请求，并分派适当的预登记的处理器来执行服务。

CORBA ORB：许多CORBA实现中的ORB核心层使用接受器 - 连接器来在客户请求ORB服务时被动地初始化服务器对象实现。[17]描述接受器 - 连接器模式怎样被用于实现The ACE ORB（TAO）[18]；TAO是CORBA一种实时实现。

WWW浏览器：像Netscape和Internet Explorer这样的WWW浏览器中的HTML解析组件使用connector组件的异步版本来建立与服务器的连接；这些服务器与HTML页面中嵌入的图像相关联。这样的行为是特别重要的，于是多个HTTP连接就可被异步地发起，以避免阻塞浏览器主事件循环。

Ericsson EOS**呼叫中心管理系统：**该系统使用接受器 - 连接器模式来使应用级呼叫中心管理器事件服务器[19]主动地与分布式中央管理系统中的被动的超级用户建立连接。

Spectrum**项目：**Spectrum项目的高速医学图像传输子系统[20]使用接受器 - 连接器模式来为存储大型医学图像被动地建立连接，并初始化应用服务。一旦连接被建立，应用就发送数兆字节的医学图像给图像仓库；或从图像仓库中进行接收。

ACE **构架：**在本论文中描述的Service Handler、Connector和Acceptor类的实现在ACE面向对象网络编程构架[6]中作为可复用组件提供。

9.11 效果

9.11.1 好处

接受器 - 连接器模式提供以下好处：

增强面向对象软件的可复用性、可移植性和可扩展性：通过使服务初始化机制与后续服务处理去耦合来实现。例如，Acceptor和Connector中的应用无关的机制是可复用的组件，它们知道怎样（1）分别主动和被动地建立连接，以及（2）一旦连接被建立，初始化相关联的Service Handler。相反，Service Handler知道怎样执行应用特有的服务处理。

这样的事务分离是通过使初始化策略与服务处理策略去耦合来完成的。因而，每种策略都可以独立地发展。用于主动初始化的策略可以只编写一次，放进类库或构架中，并通过继承、对象合成或模板实例化来复用。因而，不需要为每个应用都重写同样的主动初始化代码。相反，服务可以根据不同的应用需求进行变化。通过使用Service Handler来参数化Acceptor和Connector，可以使这样的变化的影响被局限在软件的少量组件中。

改善应用健壮性：应用健壮性是通过彻底地使Service Handler和Acceptor去耦合来改善的。这样的去耦合确保了被动模式传输端点工厂peer_acceptor_不会偶然地被用于读写数据。这消除了在使用像socket或TLI[11]这样的弱类型网络编程接口时，可能发生的一类常见错误。

9.11.2 缺点

接受器 - 连接器模式有以下缺点：

额外的间接性：与直接使用底层的网络编程接口相比较，接受器 - 连接器模式可能带来额外的间接性。但是，支持参数化类型的语言（比如C++、Ada或Eiffel）可以较小的代价代价实现这些模式，因为编译器可以内联用于实现这些模式的方法调用。

额外的复杂性：对于简单客户应用（使用单个网络编程接口与单个服务器相连，并执行单项服务）来说，该模式可能会增加不必要的复杂性。

9.12 参见

接受器 - 连接器模式使用模板方法（Template Method）和工厂方法（Factory Method）模式[10]。Acceptor的accept和Connector的connect及complete函数是模板方法，它们在连接建立时为连接到远地对端并初始化Service Handler而实现通用的服务策略。模板方法的使用使子类可以对创建、连接和启用Concrete Service Handler的特定细节进行修改。工厂方法被用于使Service Handler的创建与它的后续使用去耦合。

接受器 - 连接器模式有与客户 - 分派器 - 服务器（Client-Dispatcher-Service）模式（在[21]中描述）类似的意图。它们都关心使主动连接建立与后续服务去耦合。主要的区别是接受器 - 连接器模式同时致力于同步和异步连接的被动和主动服务初始化，而客户 - 分派器 - 服务器）模式只专注于同步连接建立。

感谢

感谢Frank Buschmann和Hans Rohnert对本论文提出的有益意见。

参考文献

[1] W. R. Stevens, *TCP/IP Illustrated, Volume* 1. Reading, Massachusetts: Addison Wesley, 1993.

[2] G. Booch, *Object Oriented Analysis and Design with Applications (2nd Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[3] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529－545, Reading, MA: Addison-Wesley, 1995.

[4] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, "Proactor - An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers," in *The 4th Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

[5] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

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

[7] W. Pree, *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1994.

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

[9] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[11] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems*,(Monterey,CA), USENIX, June 1995.

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

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

[14] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[15] D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the Ninth Edition UNIX System," *UNIX Research System Papers, Tenth Edition*, vol. 2, no. 8, pp. 523－530, 1990.

[16] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[17] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *Submitted to the IEEE Communications Magazine*, 1998.

[18] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294－324, Apr. 1998.

[19] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280－293, December 1994.

[20] G. Blaine, M. Boyd, and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System," *HIMSS, Health Care Communications*, pp. 71－81, 1994.

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