关于作者

只有原创和翻译

cuichaox@gmail.com

我的分类

- 我的文章分类
  - 设计模式
- 我的图片分类
  - 软件图的艺术
- 我的链接分类
- 我的音乐分类

反应堆模式

反应堆

对象行为类的设计模式，对同步事件分拣和派发。

1.意图

应用中，用反应堆模式处理并发的请求，这些请求可能来自多个客户端。应用提供多个服务，每个服务包含多个事件处理方法。一个服务就是一个独立的事件处理器，一个提供特定**Service**的**Handler**。事件到来时，使用**Dispatcher**（分发器）对**Handler**进行分派，这个**Dispatcher**要对所有注册的**Handler**进行维护。同时，有一个**Demultiplexer**（分拣器）对多路的同步事件进行分拣。

2.别名

Dispatcher（分发器），**Notifer**（通知器）

3.例子

为了说明反应堆模式，考虑一个例子：一个事件驱动的服务程序，这个程序在分布环境中运行，提供网络日志服务。如图**1**。客户端使用日志服务记录自己的运行情况，如：错误，跟踪调试信息，和性能报告。所有的日志记录送到日志服务器，日志服务可以把信息写入各种输出设备，如终端界面，打印机，文件，或数据库。

图**1**中，日志服务程序处理日志记录和客户端的连接请求，一个**Handle**对应一个网络连接，网络连接是被操作系统管理的资源，如套接字，不同的**Handle**上日志记录和服务请求会并发出现。

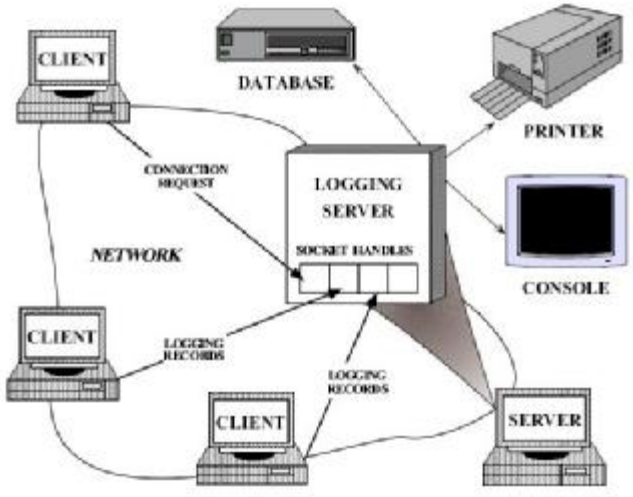


Figure 1: Distributed Logging Service

日志服务端和客户端使用面向连接的网络协议，如**TCP[1]**。在客户端发送日志以前必须先建立连接。服务端使用一个**Handle Factory**监听网络端口，等待客户端的连接请求。当连接请求达到，**Handle Factory**建立与客户端的连接，并创建一个新的**Socket Handle**代表这个连接。连接建立后，客户端就可以并发地向服务端发送日志记录。日志服务通过**Socket Handle**来接收日志记录。

开发并发的日志服务，最容易相当的方式就是多线程，使用多线程处理多个客户端的连接，如图**2**。这种方式下，为每个客户端连接创建一个独立的线程进行处理。

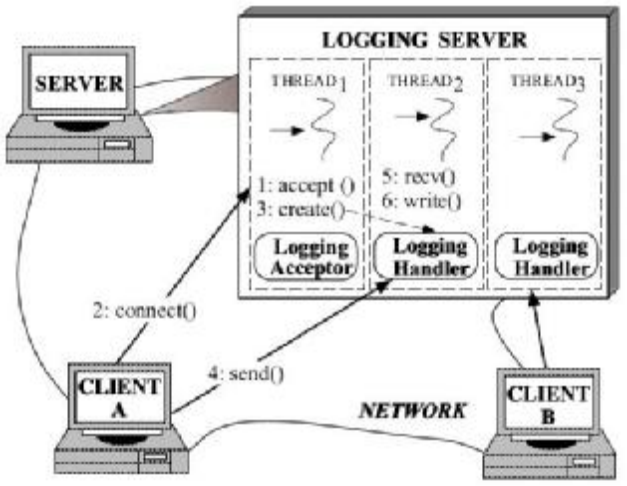


Figure 2: Multi-threaded Logging Server

但是，多线程的方式不能满足下面的要求：

性能：因为上下文切换，异步和数据迁移的消耗，多线程可能导致低效率。

编程实现的简单性：多线程要求复杂的同步控制机制。

可移植性：并不是所有的操作系统都支持多线程

因为这些缺陷，多线程常常不是最高效的方案，更不是最容易实现的方案。

4.环境

在一个分布式系统中，一个服务程序并发的接收和处理多个客户端的事件。

5.问题

服务程序必须处理多个客户端的请求。服务程序必须对到来的请求分发给正确的服务提供者。开发一个高效率的服务机构，完成分拣和派发，必须要考虑下面的需要：

可用行：服务在等待其它请求的同时处理新到来的请求。特别地，服务不能在处理请求的时候陷入无限阻塞。

效率：快速反应，高吞吐量。

编程实现简单：简化同步控制的复杂性。

适应性：当增加新的服务或者改进原有的服务，有尽量小的修改和维护成本。比如，添加一个新的服务，不需要改变分拣和分派机构。

可移植性：容易移植到新的操作系统。

6.方案

集成异步事件的分拣和派发。构造与具体应用无关的，通用的框架。

应用程序提供的每种服务对应一个独立的**Event Handler**，每种**Handler**处理一种类型的事件。所有的**Event Handler**有接口。这样，不同的**Event Handler**会注册到**Initiation Dispacher**， **Initiation Dispatcher**使用**Synchronous Event Demultiplexer**等待事件的发生。事件发生时，**Demultiplexer**通知**Dispacher**，**Dispatcher**回调**Handler**，**Handler**调用对应的事件处理方法。

7.结构

这个模式中，有下面几个关键的参与者

Handle

Handle代表操作系统管理的资源，包括：网络链接，打开的文件，计时器，同步对象等等。在我们的日志服务中，**Handle**代表与客户端连接的套接字，**Synchronous Event Demultiplexer**在这些套接字上等待事件的发生。日志服务主要关注两种类型的事件，**connection**事件和**read**事件，分别对应客户连接请求和日志记录请求。日志服务为每

一个客户端保持一个连接。每个连接就对应了一个**socket handle**。

Synchronous Event Demultiplexer

在一个**Handle**集合上等待事件的发生。这里常用系统调用**select[1]**，**UNIX**和**WIN32**平台都支持这个系统调用。**select**的返回结果说明**handle**上发生情况，需要被处理。

Initiation Dispatcher

提供接口：注册，删除和派发**Event Handler**。上面的**Synchronous Event Demultiplexer**等待事件的发生，当检测到新的事件，就把事件交给**Initiation Dispatcher**，它去回调**Event Handler**。事件种类一般有：接受到连接，数据输入，数据输出，超时。

Event Handler

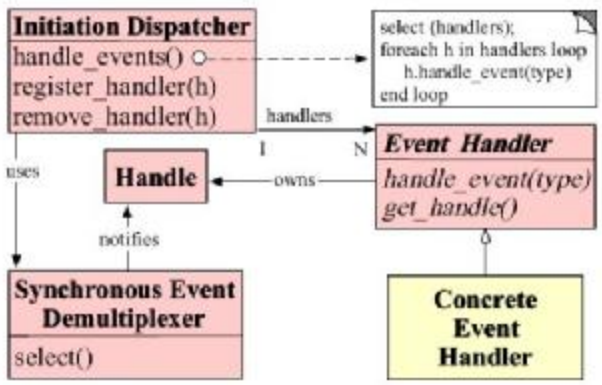
定义一个抽象接口，包含一个钩子方法，实现特定服务的派发操作。这个方法实现了与特定应用相关的服务。

Concrete Event Handler

继承上面的类，实现钩子方法。应用把**Concrete Event Handler**注册到**Initiation Dispatcher**，等待被处理的事件。当事件发生，这些方法被回调。

在日志服务中，共有两种**Concrete Event Handler** ， **Logging Handler**和**Logging Acceptor**。分别负责接受日志记录和新的客户端连接。

参考下面的类图



8.动态

8.1 概要协作过程

反应堆模式中，发生下面的协作过程

***当注册一个**Concrete Event Handler**到**Initiation Dispatcher**，要告知这个**Handler**感兴趣的事件类型。

***Initiation Dispatcher**要求每一个**Event handler**传递其内部的**Handle**。

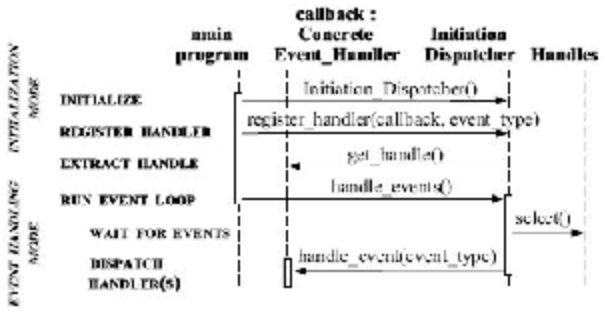
***当**Event Handler**注册完毕，进入**Initiation Dispatcher**的事件循环。**Initiation Dispatcher** 把所有**Event Handler**中的**Handle**组合在一起，使用**Synchronous Demultiplexer** 去等待事件的发生。例如，使用**select**调用等待**TCP**协议的**socket**事件。

***当事件源准备好，例如，**TCP socket**可以被读了。**Synchronous Demultiplexer**通知**Initiation Dispatcher**。

***Initiation Dispatcher**触发**Event Handler**的钩子方法。**Initiation Dispatcher**是通过**handle**定位到**Event Handler**并调用其方法的。

***Event Handler**的**hook**方法被调用完成事件的处理。

参看下图

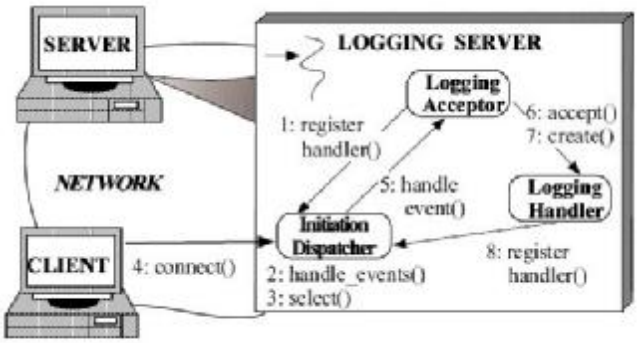


8.2 协作的场景

在日志服务，协作的过程可以分为两个场景。分别说明了日志服务如何处理客户端的连接和日志记录请求。

8.2.1 客户端连接到日志服务

在此场景中，客户端连接到日志服务，如下图。



总结为下面的步骤。

1. 注册**Logging Acceptor**到**Initiation Dispatcher**，等待处理连接请求；

2. Initiation Dispatcher的**handle_events**方法被调用；

3. Initiation Dispatcher调用**Synchronous Demultiplexer**的**select**方法，等待连接请求或日志数据的到达；

4. 一个客户端连接到日志服务器；

5. Initiation Dispatcher通知**Loging Acceptor**：有新的连接请求到达；

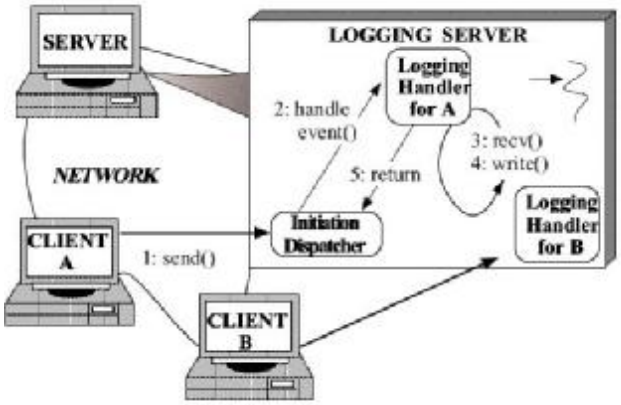
6. Logging Acceptor接受新的请求；

7. Logging Acceptor创建一个新的**Logging handler**去为新的客户服务；

8. Logging Handler把**handle**注册到**Initiation Dispatcher**。

8.2.2 客户端发送日志记录到日志服务

如下图



有下面的步骤

1. 客户端发送日志记录；

2. 当客户的日志记录数据到达套接字，**Initiation Dispatcher**通知**Logging Handler**；

3. 接受日志数据；

4. 处理日志数据；

5. 返回到**Initiation Dispatcher**的处理循环。

9 实现

这一节说明如何使用**C++**实现反应堆模式，这个实现参照**ACE**框架**[2]**的实现。

9.1 同步分拣机制

Initiation Dispatcher 利用一个**Synchronous Demultiplexer**等待一个或多个事件的发生。一般通过系统调用**select**实现。**select**用来检查哪些**Handle**上已经准备好进行**I/O**操作。一般情况下，**Synchronous Demultiplexer**都是使用操作系统提供的功能实现的。

9.2 开发一个Initiation Dispatcher

Event Handler表 ：**Initiation Disapatcher**要维护一个表，表中保存所有的**Event handler**。表中的**handler**可以在运行时添加和删除。这个数据结构有多种实现方式，哈希表，线性表，如果代表**handles**的是一个较小范围的整数，还可以使用直接索引的方法。

事件处理循环的入口点：使用**handle_events**方法提供。这个方法控制**Handle**的分拣和**Event handler**的分派。一般情况下，事件处理循环就是整个应用程序的主循环。

事件发生，**Select**调用返回，**Initiation Dispatcher**被激活，回调**Event Handler**的事件处理，事件处理完毕，又返回到**Initiatin Disaptcher**的处理中。

参看下面的C++代码

```
enum Event_Type
// = TITLE
// Types of events handled by the
// Initiation_Dispatcher.
//
// = DESCRIPTION
// These values are powers of two so
// their bits can be efficiently ‘ ‘or’ d’ ’
// together to form composite values.
{
       ACCEPT_EVENT = 01,
       READ_EVENT = 02,
       WRITE_EVENT = 04,
       TIMEOUT_EVENT = 010,
       SIGNAL_EVENT = 020,
       CLOSE_EVENT = 040
};
class Initiation_Dispatcher
// = TITLE
// Demultiplex and dispatch Event_Handlers
```

```
// in response to client requests.
{
public:
       // Register an Event_Handler of a particular
       // Event_Type (e.g., READ_EVENT, ACCEPT_EVENT,
       // etc.).
      int register_handler (Event_Handler *eh,
                            Event_Type et);

      // Remove an Event_Handler of a particular
      // Event_Type.
      int remove_handler (Event_Handler *eh,
                          Event_Type et);

      // Entry point into the reactive event loop.
      int handle_events (Time_Value *timeout = 0);
};
```

必要的同步机制： 如果反应堆模式使用单线程实现，可以不必考虑任何同步问题。

然而，在一个多线程的环境里，也可以使用**Initiation Disaptcher**作为一个事件分发中心。这时，在访问共享数据的时候，要注意同步。可以考虑使用信号量或互斥体。

为了防止死锁发生，可以考虑使用**recursive locks[4](**这个需要专门研究**)**。

9.3 派发目标的类型：

一共有两种类型的**Event handler**，可以选择其中之一，或两个都使用。

Event Handler对象。让**Handler**成为一共对象是常用的方法。在第**7**节的例子中，**Event Handler**的子类的对象被注册到**Initiation Dispatcher**。使用这种方式，很方便重用和扩展。

Event Handler函数。另外一种方式，注册函数到**Initiation Dispatcher**。增加一个处理的时候，注册回调函数，不需要增加新类。

使用**Adaptor**模式**[5],**可以同时实现两种方式。例如，一个专用的**adapter**用来封装回调的函数的指针，当这个**adapter**的**handle_event**被调用，指针指向的函数被调用。

9.4 定义事件处理的接口

假设我们使用**Event Handler**对象，下一步是定义它的接口。

单方法的接口：参看第**7**节的类图，**Event handler**基类包含一个唯一的接口函数：**handle_event**，这个函数被**Initiation Dispatcher**用来分发事件。在这个例子中，事件的类型作为事件处理方法的参数。

下面是**C++**的代码，定义一个方法接口的抽象基类。

```
class Event_Handler
// = TITLE
// Abstract base class that serves as the
// target of the Initiation_Dispatcher.
{
public:
  // Hook method that is called back by the
  // Initiation_Dispatcher to handle events.
      virtual int handle_event (Event_Type et) = 0;
  // Hook method that returns the underlying
  // I/O Handle.
      virtual Handle get_handle (void) const = 0;
};
```

单方法接口的优点，是增加新的事件处理类型，不需要改变接口。然而，使用这种方法，处理的时候要使用**switch**语句，这会影响代码的可扩展性。

多方法接口：为每一种事件类型定义一种方法。

参看下面的**C++**代码

```
class Event_Handler
{
public:
    // Hook methods that are called back by
    // the Initiation_Dispatcher to handle
    // particular types of events.
```

```
    virtual int handle_accept (void) = 0;
    virtual int handle_input (void) = 0;
    virtual int handle_output (void) = 0;
    virtual int handle_timeout (void) = 0;
    virtual int handle_close (void) = 0;
    // Hook method that returns the underlying
    // I/O Handle.
    virtual Handle get_handle (void) const = 0;
};
```

多方法接口的优点：子类重定义基类的函数，避免近一步判断分别处理。但这种方法需要框架的开发人员预先知道所有的事件处理类型。在我们的例子中，所有的**handle_***函数，都可以被**UNIX**的**select**系统调用激活。然而，这个接口不能涵盖所有**Win32 WaitForMultipleObject**机构包含的类型。

9.5 确定**Initiation Disaptcher**的个数。

大多数反应堆模式的应用，只需要一个反应堆。这种情况下，可以使用单件模式**[5]**。这种方式利于集中在一个地方处理事件的分拣和派发。

然而一些操作系统堆单个线程能够等待处理的对象进行了限制。例如，在一个线程中，**Win32**的**select**和**WaitForMultipleObject**最多能等待**64**个**handle**。这种情况下，必须数据多线程，每一个线程运行自己的反应堆。

多线程环境下，多个**Event Handler**可能同时运行，可能需要有同步机制。

9.6 **实现**Event handler

下面的代码实现了日志服务的例子，处理连接的**Logging Acceptor**和接收日志数据的**data reception**。

Logging Accesptor 类：这个例子来自**Acceptor-Connector** 模式**[8]**。这个模式分离了服务的实现和服务的激活。

Logging Acceptor 被动的接收来自客户端的连接，关键的方法和数据结构，如下面的代码

```
class Logging_Acceptor : public Event_Handler
     // = TITLE
     // Handles client connection requests.
{
public:
     // Initialize the acceptor_ endpoint and
     // register with the Initiation Dispatcher.
     Logging_Acceptor (const INET_Addr &addr);

     // Factory method that accepts a new
     // SOCK_Stream connection and creates a
     // Logging_Handler object to handle logging
     // records sent using the connection.
     virtual void handle_event (Event_Type et);

     // Get the I/O Handle (called by the
     // Initiation Dispatcher when
     // Logging_Acceptor is registered).
    virtual HANDLE get_handle (void) const
    {
         return acceptor_.get_handle ();
    }
private:
    // Socket factory that accepts client
    // connections.
    SOCK_Acceptor acceptor_;
};
```

这个类继承于**Logging_Acceptor**基类，所以可以被注册到**Initiation Dispatcher**。

Logging Acceptor包含一个**SOCK Acceptor**对象，这是一个工厂，通过这个工厂接收到新的连接请求，工厂内部维护一个监听的套接字。当连接到达，**SOCK Acceptor** 接收连接并产生一个新的**SOCK Stream**。这个**SOCK Stream**代表新的客户端连接，用来传输日志数据。

SOCK Acceptor和**SOCK Stream**其实都是**ACE**［**9]**提供的工具类。这些包装类封装了套

接字接口的细节，提供了统用的面向对象的接口。

Logging Acceptor的构造函数注册自己。

```
Logging_Acceptor::Logging_Acceptor
          (const INET_Addr &addr)
             : acceptor_ (addr)
{
    // Register acceptor with the Initiation
    // Dispatcher, which "double dispatches"
    // the Logging_Acceptor::get_handle() method
    // to obtain the HANDLE.
    Initiation_Dispatcher::instance ()->register_handler (this, ACCEPT_EVENT);
}
```

当事件到达，**handle_event**函数被调用

```
void Logging_Acceptor::handle_event (Event_Type et)
{
    // Can only be called for an ACCEPT event.
    assert (et == ACCEPT_EVENT);
    SOCK_Stream new_connection;

    // Accept the connection.
    acceptor_.accept (new_connection);

   // Create a new Logging Handler.
   Logging_Handler *handler =
             new Logging_Handler (new_connection);
}
```

这个函数调用**accept**函数被动的接收新的连接**SOCK Stream**，然后动态创建一个**Logging Handler**用来处理这个连接的上日志记录请求。如下面的说明，**Logging handler**也会把自己注册到**Initiation Distpatcher**。

Logging Handler类：这个类接收客户端发来的日志记录。

```
class Logging_Handler : public Event_Handler
// = TITLE
// Receive and process logging records
// sent by a client application.
{
 public:
   // Initialize the client stream.
   Logging_Handler (SOCK_Stream &cs);

   // Hook method that handles the reception
   // of logging records from clients.
   virtual void handle_event (Event_Type et);

   // Get the I/O Handle (called by the
   // Initiation Dispatcher when
   // Logging_Handler is registered).

virtual HANDLE get_handle (void) const
{
    return peer_stream_.get_handle ();
}

private:
    // Receives logging records from a client.
    SOCK_Stream peer_stream_;
};
```

这个类同样继承与**Event_Handler**，可以注册到**Initiation Dispatcher.** 如下说明：

```
Logging_Handler::Logging_Handler
             (SOCK_Stream &cs)
                : peer_stream_ (cs)
{
    // Register with the dispatcher for
    // READ events.
    Initiation_Dispatcher::instance ()->
    register_handler (this, READ_EVENT);
}
```

当日志数据到达，下面的函数被调用

```
void Logging_Handler::handle_event (Event_Type et)
{
    if (et == READ_EVENT) {
         Log_Record log_record;
         peer_stream_.recv ((void *) log_record, sizeof log_record);

         // Write logging record to standard output.
        log_record.write (STDOUT);
    }
   else if (et == CLOSE_EVENT) {
        peer_stream_.close ();
        delete (void *) this;
    }
}
```

和上面的处理类似，当客户端关闭了一个连接，会触发**COLSE**事件，这个事件被上面的函数同样处理。

9.7 实现服务

整个应用程序包含一个唯一的主程序。

日志服务程序的主函数：实现单线程的，并发的日志处理。事件处理循环在**Initiation Dispatcher**的**handle_events**函数里面。当用户的请求到达，对应的**Event Handler**的**hook**方法被调用，处理连接请求和日志记录请求。主函数代码如下：

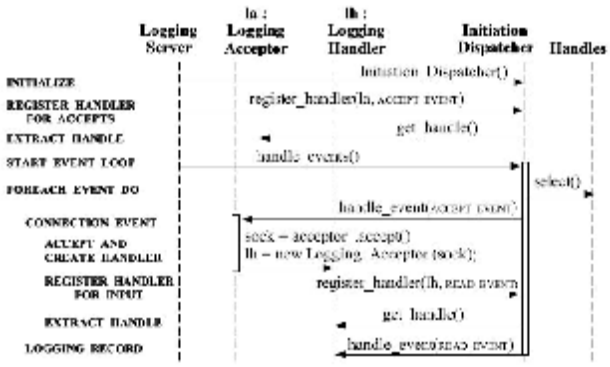
```
// Server port number.
const u_short PORT = 10000;
int main (void)
{
    // Logging server port number.
    INET_Addr server_addr (PORT);

    // Initialize logging server endpoint and
    // register with the Initiation_Dispatcher.
    Logging_Acceptor la (server_addr);

    // Main event loop that handles client
    // logging records and connection requests.
    for (;;)
        Initiation_Dispatcher::instance ()->
        handle_events ();
   /* NOTREACHED */
   return 0;
}
```

主函数创建 **Logging Acceptor**，它的构造函数会初始化并监听指定的端口。然后程序就进入了事件处理循环，在单例对象**Initiation Dispatcher**中**handle_events**会等待并处理连接和日志记录请求。

下面的交互图展示了协作的过程



一旦**Initiation Dispatcher**被实例化，这个单例的对象就成为控制的中心，在其控制下，注册其上的**Handler**的**hook**方法被触发。

连接请求到达，**Initiation Dispatcher**回调**Logging Acceptor**，**Logging Acceptor**创建一个**Logging Handler**并注册到到**Initiation Dispatcher**。然后，客户发送日志数

据，**Initiation Dispatcher**回调相应的**Logging Handler**处理日志记录，所有这些发生在一个单线程中。

10 已知的应用

反应堆模式，运行在许多面向对象的框架中。

InterViews: The Reactor pattern is implemented by the InterViews [10] window system distribution, where it is known as the Dispatcher. The InterViews Dispatcher is used to define an application′s main event loop and to manage connections to one or more physicalGUI displays.

ACE Framework: The ACE framework [11] uses the Reactor pattern as its central event demultiplexer and dispatcher.

反应堆模式在许多商业项目中使用。

CORBA ORBs: The ORB Core layer in many singlethreaded implementations of CORBA [12] (such as VisiBroker,Orbix, and TAO [13]) use the Reactor pattern to demultiplex and dispatch ORB requests to servants.

Ericsson EOS Call CenterManagement System: This system uses the Reactor pattern to manage events routed by Event Servers [14] between PBXs and supervisors in a Call Center Management system.

Project Spectrum: The high-speed medical image transfer subsystem of project Spectrum [15] uses the Reactor pattern in a medical imaging system.

11 后果

11.1 好处

使用反应堆模式，有下面的好处。

分割并征服：分离与应用无关的分发和与应用相关的处理。分发相关的组件成为可重用的组件

提高了事件驱动应用的模块化程度，可重用性和可配置性。不同的功能使用不同的类实现，例如，建立连接和接收处理数据使用不同的类。这样分离实现后，不同类型的面向连接的服务（如文件传输，远程登录或**VOD**系统）就可以重用建立连接的类。并且，当需要修改或扩展功能的时候，只需要修改**Loggning handler**类。

提高了程序的可移植性：**Initiation Dispatcher**的接口定义是与操作系统接口无关的，可以在不同的平台上实现这些接口。

提供了粗粒度的并发控制，使用单线程实现，避免了复杂的同步处理。

11.2 坏处

使用反应堆模式，导致下面的负作用。

应用受限制： 反应堆模式只能应用在支持**Handle**的操作系统上。虽然，可一使用多线程模拟反应堆，但是，因为同步控制和上下文切换的要求，这种实现效率低，与反应堆模式的出发点相违背。

非抢占模式：在单线程的实现，这种情况下，事件的处理必须不能使用阻塞的**I/O**，因此，如果存在长期操作，比如传输大量的数据。使用主动对象，效率可能更好。主动对象可以并发的处理这些任务。

难以调试

使用反应堆模式的应用程序可能会难以调试，因为程序运行的控制流会在框架和应用相关的处理器之间跳转。不了解框架的应用程序开发人员难一跟着调试。这就相调试一个词法或语法分析器生成的代码。这样的程序中，控制线程在用户定义行为的内部。当代码执行到有限自动机的内部。会难以跟踪程序的逻辑。

12 其他

和观察者模式**(Observer)[5]**相关，当一个对象改变，其他多个模块被通知。在反应堆模式中，当某个**Handle**感有兴趣的事件发生，**Handler**被通知。反应堆模式用来分拣多个事件源的事件，而观察者模式常常只关联一个事件源。

和责任链模式相关**(Chain of Responsibility)[5],**当把一个请求委托给服务提供者。反应堆模式与责任链不同，反应堆模式关联一个特定的事件处理器到特定的事件源，而责任链模式搜索一个序列去定义第一个匹配的事件处理器。

反应堆模式还被认为是异步**Proactor[18]**模式的同步变种，**Proactor**支持异步事件源的事件分拣和多个事件处理器的分派。不同的时，反应堆模式支持的是没有阻塞的同步事件源。

主动对象模式**[17]**分离了方法的执行和调用，简化了多个线程调用时同步控制的复杂性（这个模式我也翻译过了），。当多线程不可用，或者任务使用多线程太复杂了，反应堆模式经常替换主动对象。

在实现反应堆模式的时候，可以提供一个外观（**Facade**）**[5]**类处理事件的分拣。一个外观类用来隐藏一个子系统中复杂的对象关系

参考

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

[2] 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.

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

[4] D. C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit," Tech. Rep. WUCS-95-31, Washington University, St. Louis, September 1995.

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

[6] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in Proceedings of the 9th European Conference on Object-Oriented Programming, (Aarhus, Denmark), ACM, August 1995.

[7] S. Berczuk, "A Pattern for Separating Assembly and Processing," in Pattern Languages of Program Design (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[8] 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.

[9] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems, (Toronto, Canada), USENIX, June 1996.

[10] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in Proceedings of the USENIX C++ Workshop, November 1987.

[11] D. C. Schmidt, "The ACE Framework." Available from www.cs.wustl.edu/schmidt/ACE.html, 1997.

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

[13] 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.

[14] 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.

[15] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," USENIX Computing Systems, vol. 9, November/December 1996.g

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

[17] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in Proceedings of the 2nd Annual

Conference on the Pattern Languages of Programs, (Monticello, Illinois), pp. 1-7, September 1995.

[18] 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.

原文地址 http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf

发表于： **2007-02-11**，修改于： **2007-02-11 14:43** 已浏览**1013**次，有评论**0**条 推荐 投诉

网友评论

发表评论

用户名： 密码： 免费注册

验证码： E74E 匿名

提交