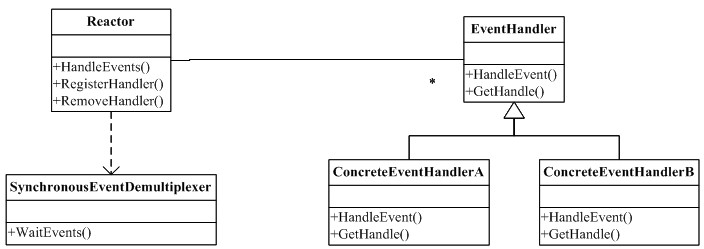
**事件处理模式之Reactor(一)**

写过大中型网络服务器的朋友相信对事件处理模型（有时也叫事件驱动模型）不陌生。今天要讲的Reactor就是在事件处理模型中用的比较多的一种设计模式。请大家先看下图，有个初步的印象：  
[](http://www.wuzesheng.com/)  
 在上图中，可以看到主要有以下四种角色：  
 1. Reactor:  
 Reactor是Reactor模式中最为关键的角色，它是该模式最终向用户提供接口的类。用户可以向Reactor注册EventHandler，然后Reactor在“反应(react)”的时候，发现用户注册的fd上有事件发生，就会回调用户提供的事件处理器。下面是一个简单的设计：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class Reactor  {  public:    /// 构造函数  Reactor();    /// 析构函数  ~Reactor();    /// 向reactor中注册关注事件evt的handler(可重入)  /// @param handler 要注册的事件处理器（不同的事件通常有不同的事件处理器）  /// @param evt 要关注的事件（读取，写入，出错）  /// @retval 0 注册成功  /// @retval -1 注册出错  int RegisterHandler(EventHandler \* handler, event\_t evt);    /// 从reactor中移除handler  /// @param handler 要移除的事件处理器  /// @retval 0 移除成功  /// @retval -1 移除出错  int RemoveHandler(EventHandler \* handler);    /// 事件循环  /// 当事件发生时处理事件,即回调注册的EventHandler中相应的事件处理函数  /// @param timeout 超时时间(毫秒)  void HandleEvents(int timeout = 0);    private:    /// reactor的实现类，不同平台下有不同的实现方式，Windows下使用select模型  ReactorImplementation \* m\_reactor\_impl;  }; |
|  |  |

2. SynchrousEventDemultiplexer:  
 SynchrousEventDemultiplexer也是Reactor中一个比较重要的角色，它是Reactor用来检测用户注册的fd上发生的事件的利器，通过Reactor得知了哪些fd上发什么了什么样的事件，然后以些为依据，来多路分发事件，回调用户的事件处理器。下面是一个简单的设计：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class EventDemultiplexer  {  public:    /// 获取有事件发生的所有句柄以及所发生的事件  /// @param events 获取的事件  /// @param timeout 超时时间  /// @retval 0 没有发生事件的句柄(超时)  /// @retval 大于0 发生事件的句柄个数  /// @retval 小于0 发生错误  virtual int WaitEvents(std::map<handle\_t , event\_t> \* events,  int timeout = 0) = 0;    /// 设置句柄handle关注evt事件  /// @retval 0 设置成功  /// @retval 小于0 设置出错  virtual int RequestEvent(handle\_t handle, event\_t evt) = 0;    /// 撤销句柄handle对事件evt的关注  /// @retval 0 撤销成功  /// @retval 小于0 撤销出错  virtual int UnrequestEvent(handle\_t handle, event\_t evt) = 0;  }; |

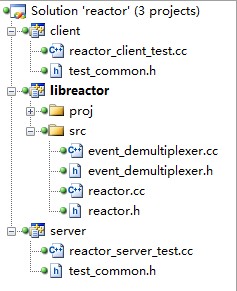
3. EventHandler:  
 EventHander是用户和Reactor打交道的工具，用户通过向Reactor注册自己的EventHandler，可以告知Reactor在特定事件发生的时候该帮我做些什么。下面是一个简单的设计：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | /// 基类  class EventHandler  {  public:    /// 获取该handler所对应的句柄  virtual handle\_t GetHandle() = 0;    /// 处理读事件的回调函数  virtual void HandleRead() {}    /// 处理写事件的回调函数  virtual void HandleWrite() {}    /// 处理出错事件的回调函数  virtual void HandleError() {}    protected:    /// 构造函数,只能子类调  EventHandler() {}    /// 析构函数,只能子类调  virtual ~EventHandler() {}  }; |

4. ConcreteEventHandler:  
 ConcreteEventHandler是EventHandler的子类，EventHandler是Reactor所用来规定接口的基类，用户自己的事件处理器都必须从EventHandler继承。

以上简单介绍了一下Reactor模式，最近我自己也在业余时间写了一个可以在windows/linux平台上运行的Reactor库，地址是：<http://code.google.com/p/xiao5geproject/source/browse/trunk/reactor/>，目前刚刚完成了库的编码，后面抽空我会加上相应的例子，以及单元测试，希望能够对写网络服务器的朋友有所帮助。

**事件处理模式之Reactor(二)**

[](http://www.wuzesheng.com/)

本文是上一篇《[事件处理模式之Reactor(一)](http://www.wuzesheng.com/?p=1607)》的下一篇，上一篇主要介绍了reactor模式的基本理论，这一篇主要通过实例，来介绍Reactor模式的具体使用。在开始本文之前，有必要再说明一下，reactor的源码的地址：[reactor源码](http://code.google.com/p/xiao5geproject/source/browse/trunk/reactor/), 感兴趣的读者朋友可以自己用svn客户端check out到本地。  
      如左图所示为reactor项目的基本结构，整体项目分为三个部分：

 一是libreactor项目，用来生成reactor相关的lib；

 二是server项目，是用reactor实现的一个时间服务器程序；

 三是client项目，是用reactor实现的一个客户端程序。

服务端和客户端之间通过telnet协议进行交互，目前仅支持time和exit两个命令，客户向服务端发送time命令，服务端返回当前时间；客户端向服务端发送exit命令，服务端断开客户端连接。如果要实现其它命令，可以直接通过扩展相关处理逻辑来实现。这里我们关注的重点是reactor模式，因此不在支持的命令方面做过多的扩展，感兴趣的朋友可以自己进行扩展。  
      下面我们来看一下server的实现，从代码中可以看出，server主要包括两个类：

 一是TimeServer类，它主要用来处理新的客户端连接请求；

 二是RequestHandler类，它主要用来处理客户端与服务端之间的命令交互。  
      接下来，我们来看一下client的实现，从代码中可以看出，client主要包括一个类：

 TimeClient类，它封装了所有客户端操作，完成与服务端之间的所有交互。  
       通过上面的介绍，以及相关源代码的学习，相信读者朋友对reactor模式的使用有了个初步的印象，下面来总结一下使用reactor的基本步骤，希望能够对用到的朋友有所帮助。总的来说，要使用reactor模式来实现一个网络server/client包括以下几个步骤：

 1. 实现处理读、写、出错事件的事件处理器EventHandler

 2. 把1中实现的EventHandler注册到Reactor中

 3. 循环调用Reactor的HandleEvents来促使Reactor进行‘反应’，不断回调EventHandler中的事件处理回调函数。